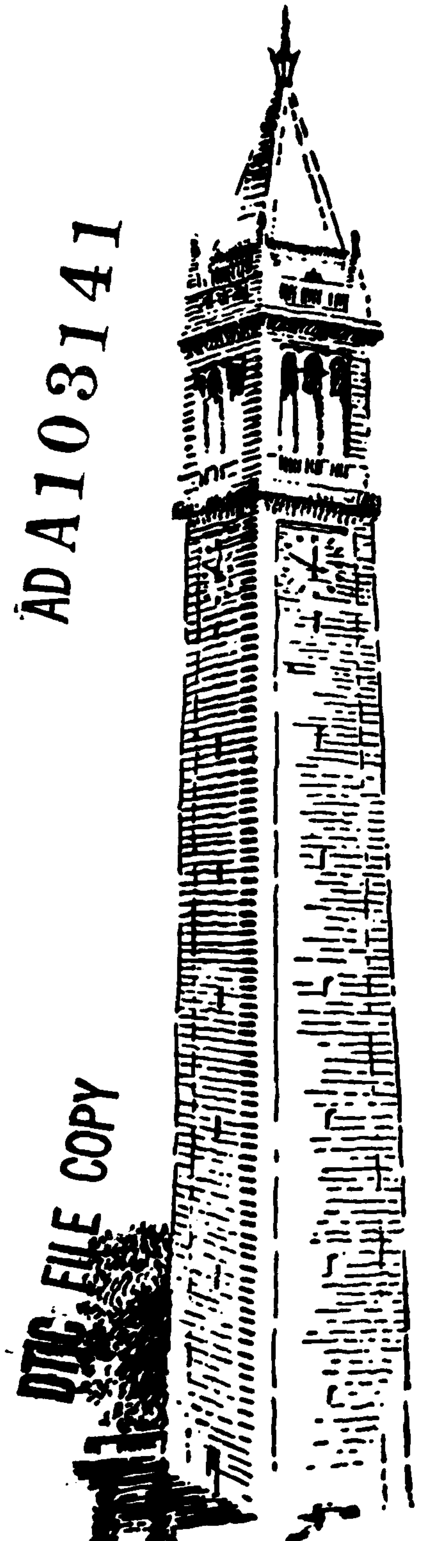
AD-A103 141   CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB   F/G 9/2
THE EFFECTS OF CONCURRENCY CONTROL ON DATABASE MANAGEMENT SYSTE--ETC(U)
APR 70 D R RIES   N00039-78-C-0013
UNCLASSIFIED UCB/ERL-M79/20   NL

1 OF 3
AD A
103141

LEVEL II

(15)



THE EFFECTS OF CONCURRENCY CONTROL ON DATABASE MANAGEMENT SYSTEM PERFORMANCE

by

Daniel R. Ries

AFOSR-78-3596

Memorandum No. UCB/ERL M79/20

April 1979

ELECTRONICS RESEARCH LABORATORY
College of Engineering
University of California, Berkeley, CA 94720

THE EFFECTS OF CONCURRENCY CONTROL ON DATABASE MANAGEMENT SYSTEM PERFORMANCE

by

Daniel R. Ries

Memorandum No. UCB/ERL-M79/20

Apr 1970

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The Effects of Concurrency Control

on

Database Management System

Performance

Ph.D. Daniel Roland Ries EECS

Signature [illegible]
Chairman of Committee

Sponsers: Air Force Office of Scientific Research
Naval Electronic Systems Command

ABSTRACT

The main goal of this thesis is to study the performance tradeoffs between parallelism and increased concurrency control overhead during simultaneous user updates of a database. During such updates, a database management system must guarantee that the multiple users do not interfere with each other.

The potential advantages of parallelism in accessing a database include the better utilization of computer resources and better response times for users. Those advantages, however, may be offset by the increased use of system resources to insure that there is no interference between the multiple users. Simulation models are used to study these two conflicting aspects of concurrency control

for both centralized and a distributed databases.

One of the most important design decisions involves locking granularity. Locking granularity refers to the size and hence the number of locks maintained by the database management system. The centralized database simulation results indicated that in many cases, in particular if data access is primarily sequential, coarse granularity such as file, relation or record type locking is preferable. However, if all of the updates are small and randomly access the database, finer granularity, such as page or record locking becomes necessary. If the sizes and access patterns of updates vary considerably, the simulation results indicated that a lock hierarchy with different sized locks is beneficial.

In a distributed database, the data is stored on different computer sites connected through some type of network. In such a system, some of the database activities are local in that they only involve data at one site. Other database activities are distributed in that they involve data at several of the computer sites. In a distributed database, increased parallelism is possible during simultaneous database activities. However, the concurrency control overhead may also increase. The simulations modeled a variety of concurrency control algorithms to study the additional tradeoffs in a distributed database.

In particular, primary site control and decentralized control algorithms were simulated. In the primary site control algorithms, one site performs the concurrency control functions for all of the other sites. In the decentralized control algorithms, the concurrency functions are distributed to each of the sites and special provisions must be used to prevent or detect deadlock.

The simulation results indicated that with a high speed network and mostly local database activities, either concurrency control approach is acceptable. As the network becomes slower, the decentralized control algorithms are preferable. If most of the database activities are distributed, however, the primary site approach can take advantage of its "global" knowledge to better schedule the processing of transactions and thus provide better performance than the decentralized algorithms.

These results can provide insights into the design and implementation of the concurrency control mechanisms for a wide variety of centralized and distributed database management systems.

ACKNOWLEDGEMENTS

I would like to thank Professor Michael Stonebraker for his guidance and support in the completion of my doctoral program. I would also like to thank Professors Lawrence Rowe and Michael Cooper for reading this dissertation. In addition thanks are owed to Karen Erdley for diligently and cheerfully typing this manuscript into the computer and to Eric Allman whose text processing support tools and expert advice made the computer production of this manuscript possible.

I would also like to thank my fellow students associated with the INGRES project for providing a stimulating and challenging environment in which to work. In addition, the support and encouragement of my colleagues at Lawrence Livermore Laboratory during the past four years is greatly appreciated.

Finally, I need to thank my family: my daughter Cindy, who claimed I was too old to go to school; my daughter Robin, who claimed that I would be a grandfather before I finished, and of course my wife Nancy, who provided support, patience and understanding.

I would also like to acknowledge the research support from the Air Force Office of Scientific Research, grant

78-3596, and the Naval Electronic Systems Command, Contract N00039-78-c-0013.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1. DATABASE CONCURRENCY CONTROL

One of the major features of a database management system is to allow multiple users access to shared data. During such multiple user access (and update), the "integrity" of the database must be guaranteed. The mechanism which guarantees that "integrity" is commonly referred to as the concurrency control subsystem of a database management system.

Two conflicting aspects of the concurrency control mechanism affect the performance of a database management system. On the one hand, the concurrency control can increase the parallelism allowed in accessing the database. On the other hand, the advantages of such increased parallelism may be offset by the amount of system resources, or overhead, that are used to insure database integrity.

The main goal of this thesis is to study the performance trade-offs between increased parallelism and increased concurrency control overhead in order to provide insights for concurrency control implementations in database management systems.

In the remainder of this chapter, some of the problems in database concurrency control are discussed and the previous research results on the performance evaluation of concurrency control mechanisms are reviewed.

2. CONCURRENCY CONTROL PROBLEMS

2.1. Database Consistency

The database concurrency control subsystem is responsible for the integrity and consistency of the database during multiple user updates. The following example illustrates the type of inconsistencies which can arise without concurrency controls.

One user is producing a summary report of the total salaries, taxes and benefits that are paid for a given pay period. At the same time some other user is updating individual payroll records for the "next" payroll. Without some type of concurrency control, the summary report may include some data from the "previous" payroll, and some from the "next" payroll. Thus, the results of that report would not accurately reflect either the previous or next payroll periods.

Furthermore, the report may not accurately reflect an individual's payroll record for either pay period. Suppose, for example, that employee x's payroll record was being updated. The summary report might contain the new salary but the old tax and benefit values.

The concept of "database consistency" refers to the permissible states of a database. The states which are permissible may require certain relationships between various elements in the database. For example, one such requirement may be that a department salary total must equal the sum of all of the individual salaries in the department. Such constraints are application dependent and thus difficult to define for a general database management system.

In [ESWA76], the concepts of transactions and serial schedules are introduced. A "transaction" is a set of related atomic actions involving a database which, if run alone on a database, preserves database consistency. A "schedule" for processing transactions is a sequence of atomic actions from the transactions. A "serial schedule" is one in which all the atomic actions from one transaction are scheduled first, followed by all of the atomic actions from a second transaction, etc. In other words, the transactions are run one at a time against the database.

A transaction schedule is "serializable" if the effects of the atomic actions in the scheduled order are equivalent to running the transactions in some serial schedule. If each transaction preserves the consistency of the database, it is clear that a serial schedule, and thus a serializable schedule, must also preserve the consistency of the database.

Two protocols for transaction behavior are defined in [ESWA76] which are used to insure the serializability of any schedule. A transaction is said to be "well-formed" if all transactions acquire a lock (read or write [DIJK68, COUR71]) before touching (reading or writing) an object of the database. A transaction is said to be "two-phased" if it acquires all of its locks before releasing any locks.

If all transactions are two-phased and well-formed, [ESWA76] shows that any schedule of atomic actions that does not violate the required locking protocols is serializable and thus preserves the consistency of the database.

Some database management systems support weaker forms of consistency where the applications may allow for certain violations of the well-formed and two-phased protocols [GRAY76]. It has also been shown [BERN78] that serialization (or effective serialization) of transactions is sometimes unnecessary. Throughout this study, however, it is assumed that the concurrency control subsystems require that transactions are well-formed and two-phased.

2.2. Deadlock and Rollback

Those two protocols do provide solutions to some of the concurrency control problems. However, other problems which the concurrency control subsystem must still solve include deadlock resolution and the problem of cascading rollback of transactions.

A simple example can be used to illustrate the deadlock problems. Suppose one transaction locks and writes object A and another transaction similarly locks and writes object B. Then, the first transaction requests a lock on B while the second transaction requests a lock on A. The four conditions for deadlock [COFF71] are met since neither transaction can release its existing locks without violating the two-phased locking protocol. Thus, a concurrency control scheme must solve deadlock problems by either prevention or detection and resolution.

If deadlock detection and resolution is used it may be necessary to roll back or undo the effects of a transaction. Note that if locking is not two-phased, some other transaction may read the effects of a transaction which has been rolled back. In this case the other transaction must also be rolled back. (Otherwise, the updates of the rolled back transaction might still appear in other parts of the database).

This condition is called "cascading" rollback and can be generated even if two phased locking is enforced. A transaction may also be rolled back because of a change in a user's mind, or because of a hardware problem. If that transaction had released some of its write locks, other transactions might also have to be rolled back. To prevent this cascading rollback, many database systems hold all locks until the end of the transaction. In fact, all of the concurrency control subsystems considered in this study will require that locks be held until the end of a transaction.

2.3. Database versus Operating System Concurrency

The concurrency control requirements for databases are different than the concurrency control requirements for operating systems. One difference is that an operating system controls simultaneous access to fixed objects; such as line printers, tape drives, specific addresses in core, etc. A database system, on the other hand, controls access to objects whose names and addresses can change.

Another difference is that more objects need to be locked in a database management system. The database may contain millions of objects, such as records, field values, etc., which have to be locked. The number of different objects that can be locked in an operating system is generally much smaller.

3. PREVIOUS PERFORMANCE RESULTS

The results of the above problems and consistency requirements have resulted in a wide variety of different concurrency control mechanisms. The goal of this thesis, however, is not to develop new concurrency control algorithms, but to study the affects of various concurrency control strategies on the overall performance of the database management system by means of simulation models. Previous work in this area can be divided into centralized databases, where the entire database is maintained by one computer; and distributed databases, where the database is distributed across several computers connected by some type of network.

3.1. Centralized Databases

In [NAKA75] a simulation model is used to study the performance of a database system. A database system model and synthetic user application models were run to estimate system utilizations and average response times. One result observed was that the system bottlenecked due to the delays caused by concurrent updates. When the concurrent updates were administratively removed from the application model (to presumably be run at night), the average response time decreased by a factor of seven. Since not all applications allow for administrative concurrency control, it is clear that concurrency control mechanisms can significantly affect the overall performance of the database management system.

Several other simulation studies have also explored the effects of concurrency control on database system performance. In [SPIT76] the effect of scheduling the lock requests and releases for the System 2000 database management system was examined. In that study, the difference between locking the database for the entire period of a transaction, as opposed to locking and unlocking the database for each atomic update was surprisingly small. The additional parallelism possible with the short locks was offset by the additional time spent by the transactions waiting for that lock.

In [MUN77] several parameters and concurrency control alternatives were explored by means of a simulation model. In that simulation, alternate methods for choosing a victim in deadlock resolution were explored. The results of the simulation showed that three methods for selecting a victim were superior: 1) the victim should be the process which accessed the least amount of data, 2) the victim should be the process which held the fewest number of locks; or 3) the victim should be the process which had used the least amount of computer resources.

In addition to deadlock resolution, the [MUN77] simulation was used to study the optimum number and size of the lockable data units in the database. The authors concluded that the units of locking should be very small. However, that conclusion was not based on a fixed application environment. Instead, the sizes of the transaction were made smaller as the sizes of the locks were reduced. Thus whether the observed increase in parallelism was due to the smaller transactions or the smaller locks is unclear. Two other problems with that study were that only the CPU utilization was considered and that the CPU resources of their model were effectively considered infinite.

In Chapter 2, a simulation model is used to further study locking granularity, optimum lock duration and a variety of other factors.

3.2. Distributed Databases

Recently, considerable attention has been devoted to the development and use of distributed databases [LBL76, LBL77, LBL78]. In such an environment, the data is distributed across a network of computer systems. The potential benefits of such distribution include sharing of data across different computer sites, increased parallelism in accessing the database, locating data closer to users and increased reliability.

However, one of the major problems with a distributed database is the development of a concurrency control scheme to insure database consistency during multiple user updates [STON77]. Concurrency control schemes for a centralized database do not always extend to a distributed database.

For example, in a centralized database, a transaction can request all of its locks at the beginning of its processing and release them at the end [CHAM74]. In this scenario, the locks are acquired in one atomic action. If one lock is denied, all locks are denied and the entire lock acquisition step is repeated. Note that in this scenario, deadlock is impossible.

In a distributed database, however, locks may have to be obtained at distinct computer sites. Even though the lock acquisition at each site is atomic, deadlock can still occur because one processing unit does not access the entire database. Concurrency control considerations require that the different processing units communicate with each other. The communication must be used either to centralize the concurrency control functions or to prevent or detect a decentralized deadlock.

Several solutions to the concurrency control problems for distributed databases have been proposed [BERN77, ROSE77, GRAY78, MENA78 and STON78]. To evaluate the performance of the different proposals, the number of messages which must be sent for concurrency control are counted. In [BERN77] it is shown that if the transactions are known in advance (i.e. only certain known types of transactions access the database), different types of concurrency control can be used for different types of transactions and thereby further reduce the network concurrency control traffic.

Unfortunately, a count of overhead message traffic does not, by itself, determine the effects of the concurrency control on the overall performance of the distributed database system. Other factors such as overall system load, the amount of non-local processing, and the scheduling of transactions must also be considered.

In Chapter 3, a simulation model is used to examine the effects of these factors as well as the effects of the message traffic.

In a distributed database, the same data may be stored at several computer sites. These multiple copies create additional concurrency control problems in that the copies must be kept mutually consistent during multiple updates. (The simulations in chapter 3 do not explicitly model the multiple copy update scenarios. However, some of the results of the study can be applied to the multiple copy update problems.)

Other studies do directly model the multiple copy update problems but do not address the internal database consistency issues. In [GRAP76], different algorithms for multiple copy consistency are analysed in terms of the performance of a distributed database management system. In [GARC78], a simulation model is used to compare the effects of two algorithms [ALSB76, THOM78] on the overall performance of the distributed database system. Both studies show that under a wide variety of assumptions a "primary copy model" is better for maintaining multiple copy consistency. The primary site model basically implies that the control of updates to the different copies is channelled through a single or primary copy of the database.

4. OVERVIEW

This thesis will analyze the effects of concurrency control on the performance of both centralized and distributed databases. In both cases, simulation models are used to study the tradeoffs between increased parallelism and increased locking overhead.

One parameter of primary interest is the locking granularity. Locking granularity refers to the size of a lockable unit or granule which covers a portion of the database. Locking granularity can be extremely fine (i.e. one lock is associated with each sector of a disk). Or, locking granularity could be extremely coarse (i.e. one lock is associated with each disk drive).

In Chapter 2, an extensive simulation model is presented which explores a large class of concurrency control alternatives. The model is parameterized to provide insights into the locking parameters for a wide variety of database systems. The simulation experiments study locking granularity, the overhead costs of locking, the transaction types and sizes, a locking hierarchy, and the times when locks are acquired. Most of these results have been published previously [RIES77, RIES79].

In Chapter 3, the simulation models are extended to distributed database systems. These experiments study the effects on performance of locking granularity, four distributed concurrency control algorithms, the transaction types and sizes, and various network related parameters.

In Chapter 4, the major results these studies are summarized and several directions for future research are suggested.

CHAPTER 2

CENTRALIZED DATABASE SYSTEMS

1. INTRODUCTION

In order to insure the consistency conditions discussed in chapter 1, a variety of concurrency control mechanisms [CHAM74, CODA71, ESWA76, GRAY75, GRAY76, MACR76, STEA76, STON74] have been proposed and implemented in single machine database management systems. In this chapter the performance issues of these types of mechanisms are examined.

Clearly there are advantages to increasing the parallelism in processing transactions. Unfortunately, the price of this increased parallelism is the increased overhead which must be expended to achieve it. The goal of this chapter is to study the tradeoff between these conflicting performance considerations on a single machine database management system.

In section 2, a simulation model is developed to study that tradeoff. In section 3, the results of experiments with that simulation model are reported. In addition, two extensions to that model are used to study alternate concurrency control mechanisms. Finally, the major conclusions are summarized in section 4. In the remainder of this section, the performance issues and approaches of centralized concurrency control are reviewed.

1.1. Performance Issues

An evaluation of concurrency control must include an analysis of the tradeoffs between the overhead spent on locking versus the advantages of allowing more parallel access to the database. The advantages of increased parallelisms are in terms of better user response time and increased machine utilization.

The amount of overhead spent on locking is dependent on several parameters of the concurrency control mechanism. These parameters include the size of a lockable unit, the costs of setting and releasing locks, the proportion of resources required for locking and the length of time for which the locks are held. Each of these parameters is considered in turn.

1.1.1. Locking Granularity

All of the concurrency control mechanisms involve the locking of some physical or logical portion of the database. The smallest portion of the database which can be locked is referred to as a "granule". The size of a granule varies in different database management systems. In some systems (CODASYL [CODA73], System R [ASTR76], DMS-1100 [GRAY75]) the granule may be as small as one record. Other systems (System 2000 [SPIT76], IMAGE [HEWL77]) support one granule covering the entire database. Still other systems (DBMS-11 [DEC77], LSL [LIPS76]) support intermediate sized granules such as files or areas.

There is a clear tradeoff between locking overhead and parallelism based on the locking granularity. Fine granularity allows a higher degree of parallelism at greater cost in managing locks. For example, assume that a granule corresponds to a record in a database. Then the transactions may run in parallel without conflict as long as they access distinct records. However, the database system must be prepared to handle as many locks as there are records in the database.

Coarse granularity, on the other hand, inhibits parallelism but minimizes management of locks. If the granule is considered the entire database, no transactions will run in parallel. The database system keeps track of only one lock.

Different sized granules can be supported in a lock hierarchy [GRAY76]. In a lock hierarchy, a tree structure of locks is supported. A transaction can either explicitly hold locks on lower branches of the tree, or implicitly hold those locks by explicitly locking an ancestor node common to all of the lower branches.

With such a hierarchy, the costs of locking for large transactions may be greatly reduced since it is cheaper to set one large lock than to set many small locks. However, the locking costs for the transactions using small locks may increase. Those transactions would have to set all the locks in the path from the leaves to the root of the tree. Again a tradeoff is observed between the parallelism allowed and the locking overhead.

For example, consider a two-level hierarchy where one lock at the top level controls access to the entire database and many (more than one) locks at the lower level control access to individual "parts" of the database. A transaction which accesses the entire database can exclusively lock the one top level lock. Without a lock hierarchy and just the small locks, it would be much more expensive for that transaction to lock all of the small locks.

The above lock hierarchy, however, increases the locking overhead for the transaction which just access one "part" of the database. That transaction must set the higher-level lock in an "intention" mode [GRAY76] (implying that explicit locking is required at the lower level) and still lock the individual part of the database. Thus, that transaction sets two locks. Without a lock hierarchy and just the small lock, that transaction would set only one lock.

1.1.2. Locking Overhead

Concurrency control overhead refers to the amount of computer resources utilized by the locking mechanism. This "overhead" can be thought of as the difference between the resources required by a transaction in a multi-user system and the resources that would be required if the transaction could be run as the only user of the database. The locking mechanisms must compete with the transactions for memory, CPU cycles, and I/O channels. Thus, as a locking mechanism increases in complexity and requires more resources, it will start to interfere with the running of the transactions.

The ratio of resources spent for locking to resources spent for processing transactions is critical. For example, a ratio of one implies that it is as expensive to lock a granule as it is to process the data in that granule. In this case, two transactions could have been run without locking in the time it takes a transaction to set its locks, process the data, and release the locks. A less expensive concurrency control which only allowed half of the parallelism might provide the same throughput.

1.1.3. Lock Duration

Another factor which affects the degree of parallelism and the cost of concurrency control is the time period for which the locks are held. Two simple procedures which insure that a transaction is two-phased (See Chapter 1) are:

1) set all locks at the beginning of a transaction or

2) hold all the locks until the end of a transaction.

If the second option is chosen, the locks can still be preclaimed as in option 1 or requested and granted as needed by the transaction.

A performance tradeoff is again possible. By not locking resources until they are required, additional parallelism is possible. However, the locking overhead costs are increased by two factors. First, the concurrency control mechanism must check for deadlock [COFF71]. Second, if deadlock is detected, a transaction may have to be restarted. The resources already used by a restarted transaction should also be included as overhead costs since they would not have been used if the transaction was run by itself.

In summary, the important performance parameters are locking granularity, locking overhead, and lock duration.

1.2. Locking Mechanisms

Two general options have been proposed for single machine concurrency control -- physical locks and predicate locks.

1.2.1. Physical Locks

Physical locks are placed on records, pages, segments, files, areas or the entire database. In this case, a "granule" refers to a physical portion of the database.

With physical locks, a data manipulation command cannot proceed if a granule it needs is locked by someone else. Various strategies for requesting and releasing locks have been suggested [CHAM74, GRAY76, STEA76, MACR76]. Some of these strategies require the detection resolution of deadlock.

The basic idea behind physical locking is straight forward. If a transaction needs to read a portion of the database but not write it, a read or shared lock must first be obtained on a granule which physically covers that portion of the database. Other transactions which also read that portion or a portion covered by the same granule can share that lock.

If all or a portion of the granule is to be updated, an exclusive lock must be obtained which cannot be shared by other running transactions. The two-phase requirement insists that no locks can be released until the transaction has acquired all of the locks that it needs.

1.2.2. Predicate Locks

A predicate lock can be set on the exact portion of the database which is to be accessed. The portion of the database which is locked is determined by predicates or qualifications associated with the transaction. The predicate (i.e. "all records with date field values in June, 1976") restricts the transaction to a logical subset of the database. Such locks do not necessarily correspond to any physical portion of the database. This approach is explored in [FLOR74, STON74, ESWA76].

In predicate locking, a predicate corresponding to a selection criteria of a transaction is submitted to the locking mechanism. If the locking mechanism can "prove" that a transaction does not conflict with any running transactions, the locks are granted. Otherwise, the requesting transaction must wait. A propositional logic "theorem prover" can be used to prove that two transactions do not conflict. The sophistication of the theorem prover can be varied depending on the tradeoff between increased parallelism and CPU recourses used for locking.

The granularity in predicate locking also varies. For example, a predicate such as "employee_no=1234" might restrict the transaction to one record. On the other hand, a predicate such as "department=engineering" might represent hundreds of records. Notice that predicate locks, like physical locks, can be acquired throughout the duration of a transaction.

Predicate locking has two obvious advantages. One is that a predicate lock can accurately describe the exact logical portion of the database that is to be accessed by a transaction. The second is that the cost of setting such a lock depends on the number of simultaneous transactions actually submitted and not on the amount of data that is actually accessed.

However, the predicate locking mechanism may needlessly keep a transaction from running. Suppose the predicate "AGE>29" has been granted a lock for a running transaction and that another transaction issues the request "AGE<31". If no one with AGE=30 were in the database, both transactions could be allowed to run. But the predicate locking mechanism would require that the second transaction waits.

Thus, predicate locking may reduce concurrency control overhead at the expense of allowing less parallelism in accessing the database.

2. MODEL DESCRIPTION

A simulation model is used to investigate the tradeoffs between concurrency control overhead and increased parallelism. The model is described by first explaining the flow of transactions around a closed-loop model. Next the model input and output parameters are discussed. Finally, the allocation and competition for resources in the model are described.

2.1. Transaction Flow

The running of transactions against a database is simulated by assuming there are a fixed number of transactions which are cycled continuously for TMAX time units around the model shown in figure 2-1. A transaction goes through the following steps:

1) Arrive on the pending queue

2) Acquire locks

3) Process I/O requests

4) Process CPU requests

5) Release locks

6) Generate a new transaction and return to step 1

These steps are explained in more detail below.

Initially, the transactions arrive one time unit apart and are placed on the pending queue. The transaction is removed from the PENDING queue and all required locks are requested. If the locks are granted, the transaction is placed on the bottom of the I/O queue. If the locks are denied, the transaction is placed on the bottom of a BLOCKED queue. The blocking transaction is recorded. (The description of which locks are required by a

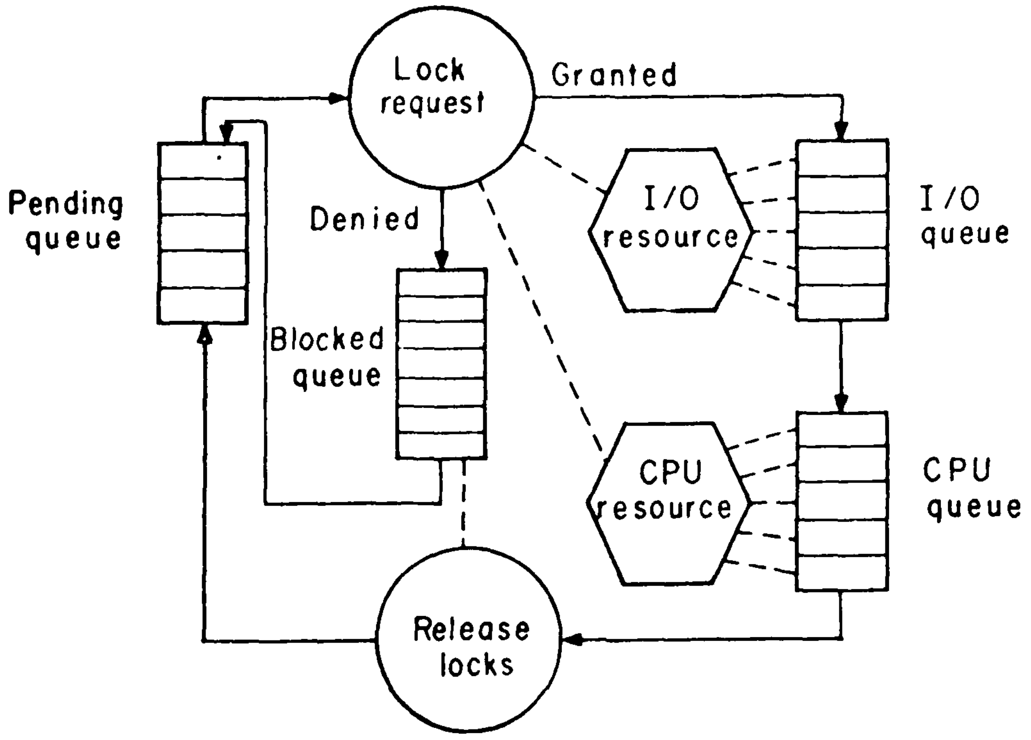


Figure 2-1 Simulation model

transaction is given in section 2.3.) Note that no locks are held while on the blocked queue so deadlock is impossible.

After completing the I/O required, the transaction is placed on the bottom of the CPU queue.

After completing the CPU required, the transaction releases its locks. At this point a new transaction is added to the end of the PENDING queue. (Note that each transaction goes through one I/O phase and one CPU phase. Although they are sequential in the model, the result would be the same if each transaction were to go through many I/O - CPU phases in a single cycle.) All transactions that were blocked by the completed transaction are placed on the front of the PENDING queue.

2.2. Model Parameters

The input parameters can be divided into those that characterize the workload, and those that characterize the system. Workload parameters describe the database and the transactions that are run against that database. System parameters describe the computer system and/or the database management system characteristics. The output parameters characterize the throughput, overhead and utilization of the system. These parameters are all described in more detail below.

2.2.1. Workload Parameters

The workload input parameters are summarized in Table 2-1. The number of transactions, NTRAN, in the system is fixed. The closed loop model could have two interpretations. As each transaction completes, the user submits another transaction. Alternately, the transactions could be viewed as application programs. When one of these completes, another application program enters the system to take its place.

The database size, DBSIZE, refers to the number of entities in the database. In this model, the database is an abstract collection of entities. An entity can be

Table 2-1 Workload Parameters

Parameter	Description
NTRAN	number of transactions
DBSIZE	number of entities in the database
RAD	a transaction size parameter.
AMEAN	mean for exponential distribution of transaction size.
BMEAN	another mean for exponential distribution of transaction size. Used with AMEAN for hyper-exponential distribution of transaction sizes.
ALPH	cut of proportion between AMEAN and BMEAN.
LKPLMT	lock placement assumption (see below)

thought of as the unit of data moved by the operating system into the database system buffers.

In the simulation, three types of distributions for transaction sizes are used. The size of a transaction refers to the number of entities, NE, touched or accessed by a given transaction. The number of entities "touched" or accessed by a given transaction completely determines the amount of I/O, CPU and lock resources required by that transaction. In the simplest case, the sizes of the transactions are uniformly distributed by the RAD parameter. The number of entities touched by the ith transaction is given by:

$$NE_i = i*RAD, for i = 1, ..., NTRAN$$

This distribution reflects a workload with a uniform mix of different sized transactions.

The second distribution of transaction sizes is exponential. The average transaction size is determined by the AMEAN parameter. In this case,

$$NE_i = -AMEAN*\log(rnd)$$

where rnd is a uniformly distributed random number between zero and one. This distribution reflects a workload where most of the transactions are small a very few transactions are large.

The final distribution used is hyper-exponential with three parameters, AMEAN, BMEAN, and ALPH. In this distribution, some (ALPH x 100 percent) of the transactions have sizes which are exponentially distributed with a mean of BMEAN. The other transactions have sizes which are exponentially distributed with mean AMEAN. In this case,

$$NE_i = -BMEAN*\log(rnd1)$$

if rnd2 < ALPH or

$$NE_i = -AMEAN*\log(rnd1)$$

if rnd2 >= ALPH

where rnd1, and rnd2 are independent random variables similar to rnd. This distribution is used to model scenarios where, for instance, most of the transactions are extremely small and only touch a few records or pages of a database, while a few transactions must access a very large number of records. Note that the exponential distribution is just a special case of the hyper-exponential distribution with an ALPH of zero.

The lock placement parameter, LKPLMT, determines the number of locks that a given transaction requires. Three different assumptions regarding lock placement are simulated.

With "well placed" locks (LKPLMT = 1), the number of locks required by a given transaction is exactly proportional to the percentage of the database touched or accessed by the transaction. For transaction i, the number of locks, NL, is

$$NL_i = \mathrm{CEILING}(NE_i * NGRAN/DBSIZE)$$

where NGRAN is the total number of locks available. Hence, a transaction which touched half of the entities in the database would require half of the possible database locks. Note that this amounts to assuming that the granules are "well placed", i.e. that the entities needed by the transactions are packed into as few 'lockable' granules as possible. This assumption is reasonable for transactions which access the database sequentially. Although sequential processing in database applications has been observed [RODR76], actual transactions may require a combination of sequential and random accesses to the database.

Under a "worst case placement" assumption (LKPLMT = 2), each transaction requires the maximum number of granules possible. In this case

$$NL_i = \mathrm{MIN}(NE_i, NGRAN).$$

If the total number of entities touched by a given transaction, NE, is greater than the number of locks covering the entire database, NGRAN, then in the worst case, all of the locks might have to be acquired in order to access the needed entities. If NE is less than NGRAN, the number of locks that have to be set is bounded by the number of entities, NE. Thus, the number of locks required is the minimum of the number of locks for the entire database and the number of entities touched by the transaction. This assumption simulates an "uncooperative" transaction, i.e. one whose access pattern is the worst possible from the lock mechanism point of view. This scenario is the opposite extreme of the "well placed" assumption.

Under a "random access placement" assumption, a mean-valued formula is used to estimate the number of locks required for each transaction. The number of locks required under this assumption is analogous to the number of blocks accessed when randomly selecting records from a blocked file. The formula for this number and its derivation are given in [YAO77]. This model accurately reflects random processing where the probability of accessing any entity is the same and independent of any previous entities accessed. Let DBSIZE be the number of entities in the database, NGRAN be the total number of locks, and p be the number of entities per lock (=DBSIZE/NGRAN). Then a transaction which accesses NE entities requires

$$NGRAN * \left[1 - \frac{C_{NE}^{DBSIZE-p}}{C_{NE}^{DBSIZE}} \right]$$

locks. The expressions $C_{NE}^{DBSIZE-p}$ and C_{NE}^{DBSIZE} represent the number of different ways NE entities can be selected from DBSIZE-p and DBSIZE entities respectively. The derivation of this formula is identical to the derivation in [YAO77] and is not repeated here.

2.2.2. System Parameters

The system parameters are listed in Table 2-2. The number of granules, NGRAN, into which the database is divided is varied from one to the number of entities in the database, DBSIZE. A granule is the unit which is locked by a transaction. Each granule is assumed to be the same size. Hence, if NGRAN = 1, a granule is the entire database of DBSIZE entities. If NGRAN = 2, a granule is DBSIZE/2 entities. If NGRAN = DBSIZE, each granule is 1 entity.

The CPU costs for processing a transaction are determined by the CPURATE parameter. The CPURATE refers to the CPU resources required for processing one entity of the database where CPU resources are in time units of the

Table 2-2 System Parameters

Parameter	Description
NGRAN	number of lockable granules
CPURATE	CPU time to process one entity
IORATE	I/O time to process one entity
LCPURATE	CPU rate to process one lock
LIORATE	I/O rate to process one lock
IOOVLP	number of simultaneous I/O operations permitted

simulation. Note that these are the resources for processing the transactions and do not include any costs for processing the locks.

Similarly, the I/O costs for processing a transaction are determined by the IORATE parameter. Note that the CPURATE and IORATE could have also been considered as workload parameters because in many cases they are application dependent [HAWT79].

The lock CPU parameter, LCPURATE, refers to the CPU resources required to request (and set/release) a lock for one granule.

Similarly, LIORATE determines the I/O overhead for setting one lock. For some DBMS systems, lock tables are kept entirely in main memory. These systems are modeled by a LIORATE of zero. On the other hand, a database system which has as many locks as pages in the database, may have to keep lock tables on secondary storage devices.

Such systems would have a non-zero LIORATE parameter.

The I/O overlap, IOOVLP, indicates how many simultaneous I/O operations are possible. This parameter is a surrogate for the number of independent paths used between main memory and secondary storage (and hence for how much I/O activity can go on in parallel).

2.2.3. Output Parameters

The performance measurements shown in Table 2-3 are generated by each simulation run. The total CPU time, TCPU, refers to the number of time units in which the CPU is busy. During TMAX - TCPU time units the CPU is idle.

The total I/O units, TIO, is the number of time units in which the I/O resources are busy. The total I/O units utilized can become larger than TMAX if the I/O overlap

Table 2-3 Output Parameters

Parameter	Description
TCPU	Total time CPU active
TIO	Total time I/O active
LOCKCPU	CPU overhead for locking
LOCKIO	I/O overhead for locking
USEFULCPU	CPU time for transactions
USEFULIO	I/O time for transactions
TRANCOM	number of transactions completed
AVERRES	average response time

parameter is greater than 1. In fact TIO is bounded above by TMAX * IOOVLP.

The CPU units used for lock management are recorded in LOCKCPU while the I/O units used for locking are recorded in LOCKIO.

The useful computer utilization, USEFULCPU and USEFULIO, refer to the net resources used for transaction processing. These measurements reflect the transaction processing time without the concurrency control overhead. Note that

$$TCPU=LOCKCPU+USEFULCPU$$

$$TIO=LOCKIO+LOCKCPU.$$

The total number of transactions completed at the end of a simulation run, TRANCOM, and the average response time, AVERRES, are also recorded. The time when each transaction is first placed on the pending queue is considered an arrival time for that transaction. The difference between that time and the time when that transaction releases its locks is called the response time. Some transactions may have started but not finished at the completion of the simulation run. These transactions are not included in the computation of TRANCOM or AVERRES.

2.3. Resource Allocation and Usage

The above parameters completely determine the resources required by each transaction. These resources are summarized in Table 2-4.

The CPUTIME represents the total number of time units that a transaction would be on the CPU queue if it were running by itself. However, if there are N transactions on the CPU queue, the CPU is multiplexed among those N transactions. For example, if there are always 2 transactions on the CPU queue, a transaction with a CPUTIME = 50, would remain on the CPU queue for 100 time intervals.

The IOTIME is similar, except for the effect of the IOOVLP parameter. If there are N transactions on the I/O queue, each transaction progresses min(1,IOOVLP/N) time

Table 2-4 A Transaction

RESOURCE	FORMULA
NE	= function(RAD) or function(AMEAN,BMEAN,ALPH)
NL	= function(NE, LKPLMT, DBSIZE)
CPUTIME	= NE * CPURATE
IOTIME	= NE * IORATE
LOCKIOTIME	= NL * LIORATE
LOCKCPUTIME	= NL * LCPURATE

units. The progress is bounded above by 1 to simulate one transaction having only one outstanding I/O request at a time.

The locking mechanisms are given a higher priority for the I/O and CPU resources than the active transactions. Also note that these costs are repeated each time a transaction requests its locks. For example, suppose a transaction requests locks, they are denied, and the transaction is placed on the blocked queue. Later that transaction is removed from the blocked queue, the locks are requested again, and this time they are granted. The total lock overhead associated with this transaction is twice NE times the lock rates.

Two approaches are used to simulate the competition for the available granules. Under both approaches, the decision to grant or deny a lock request is based on another uniformly distributed random variable, rnd3.

Under one approach, the granules for each transaction are considered to be completely uncorrelated. Let CRL be the number of locks currently held by the active transactions. Then a transaction needing NL locks, has those locks granted if

$$rnd3 > \frac{C_{NL}^{NGRAN-CRL}}{C_{NL}^{NGRAN}}$$

The above expression is simply the number of ways of choosing NL locks from those that are already still unclaimed divided by the number of ways of chosen the NL locks from all of the locks.

Under the well-placed lock assumption, the above formula actually penalizes finer granularity. For example, doubling the number of locks, (2 * NGRAN), could result in also doubling NL and CRL. The number of locks for a transaction, NL, would be doubled if a transaction touched all of the entities covered by a given lock. But then, the probability of a successful lock request is actually smaller due to the finer granularity because

$$\frac{C_{2*NE}^{2*NGRAN-2*CRL}}{C_{2*NE}^{2*NGRAN}} > \frac{C_{NE}^{NGRAN-CRL}}{C_{NE}^{NGRAN}}.$$

The right hand side is the probability of obtaining NL locks with the original granularity while the left hand term is the same probability if the number of locks were doubled.

To avoid this bias under the well-placed lock assumtion, a second approach to computing lock conflicts is used. With this approach it is assumed that the first requested granule is uncorrelated with any of the granules which are already locked. Furthermore, the additional requested granules are assumed to be distinct from the already locked granules. Under this assumption, the locks are granted if

$$rnd3 > \frac{CRL}{(NGRAN-NE+1)}.$$

Under either approach, if the locks are granted, CRL is incremented by NL. If the locks are denied, one of the active transactions is picked as the blocking transaction. The probability that a transaction, say T_j is the blocking transaction is NL_j/CRL; i.e. is directly proportional to the number of locks held by the blocking transaction.

3. RESULTS and DISCUSSION

In this section the results of running the simulation under a wide variety of parameter settings are reported. First, the results of some initial runs of the simulation are explained. Next the effects of varying the workload and system parameters are reported. Finally, the effects of two changes to the basic simulation model are described.

3.1. An Initial Scenario

The simulation is initially run with the workload parameters shown in Table 2-5 for 10,000 (TMAX) time units. The system parameters for the first run are shown in Table 2-6.

In this scenario, ten transactions were submitted to a database of 5000 entities. The transactions required from 50 to 500 entities each (initially the sizes were uniformly distributed). (The simulation was also run with up to 20 transactions with no appreciable effect other than scale on the output parameters.)

Table 2-5 Sample Workload Parameters

Parameter	Value
NTRAN	10
DBSIZE	5000
LKPLMT	Well-Placed
RAD	50

Table 2-6 Sample System Parameters

Parameter	Value
NGRAN	1 to 5000
CPURATE	.05
IORATE	.20
LCPURATE	.01
LIORATE	.20
IOOVLP	1

The locks were assumed to be "well-placed" with respect to the accessing transactions and thus the transactions required the smallest number of granules that were required to "cover" the touched entities.

The I/O overlap parameter was set to one which results in only one transaction processing an I/O operation at one time. Note that for this run the I/O rate is four times the CPU so that this simulates an "I/O bound" application. The CPU cost of a lock was 1/5 that required to process an entity. Lastly, the I/O cost of a lock was equal to the I/O cost of an entity. Hence, this initial run simulated a lock table being kept on secondary storage.

Intuitively, these input parameters could be interpreted as followes:

DBSIZE is 5 million bytes (one entity is 1024 bytes)

Average transaction accesses 250,000 bytes of data.

IORATE of 30 msecs per entity (one disk accesses).

CPURATE of 7.5 msec per entity.

LIORATE of 30 msecs per lock.

LCPURATE of 1.5 msecs per lock.

In this interpretation one time unit corresponds to 150 milliseconds.

For these simulation runs, the value TMAX = 10000 was chosen after running simulations for various smaller values including TMAX = 2500. In all cases, no change (except for scaling) was observed in the output parameters between TMAX = 2500 and TMAX = 10000. For some of the experiments discussed later, other values of TMAX were required to guarantee convergence. Keeping the other parameters fixed, the number of granules was varied between 1 and 5000. The output from the simulations is presented in Tables 2-7 and 2-8.

Note that the utilization of I/O resources for transaction processing, USEFULIO, peaked at 40 granules. Within 1% of this value was reached with only 10 locks. The useful I/O remained relatively constant until the lock I/O costs start to be a significant fraction of I/O time. For a small number of granules, high lock I/O cost resulted from lock conflicts which generated still more lock I/O. (In an actual implementation of a locking scheme, a small number of locks could easily be maintained in primary memory. This alternative is explored subsequently.) Similarly, the useful CPU time peaked at 30 granules, and again this value was almost reached (within 1%) with as few as 10 granules. These results are portrayed graphically in figure 2-2. The lock CPU costs were

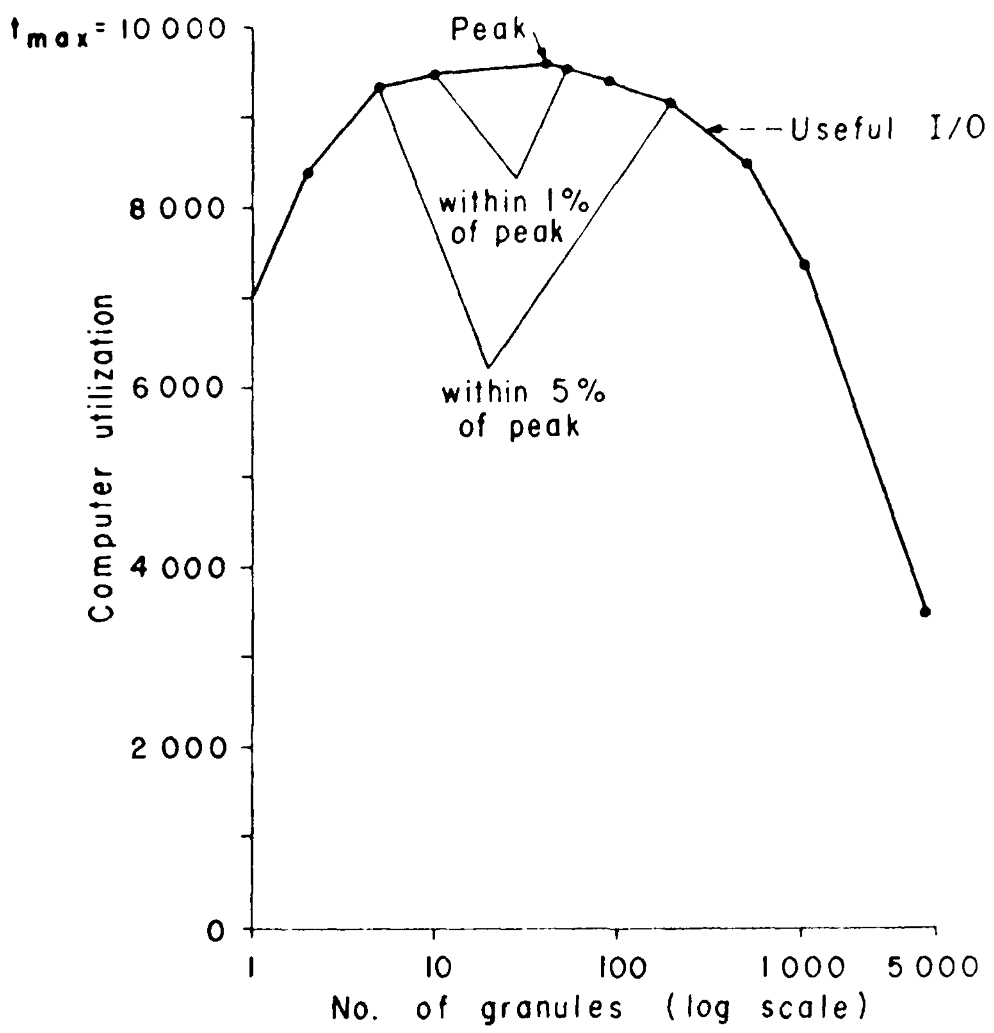


Figure 2-2 Computer utilization versus no. of granules.
Initial scenario, Well-placed locks

Table 2-7
CPU and I/O Utilization
Initial Scenario

NO_of_GRANULES	USEFULIO	USEFULCPU	LOCKIO	LOCKCPU
1	7041.957	1759.906	1282.000	12.820
2	8376.933	2091.914	970.000	9.700
3	9002.256	2237.415	777.000	7.770
4	9030.253	2258.925	671.000	6.710
5	9273.915	2304.927	604.000	6.040
7	9438.514	2309.940	474.000	4.740
9	9449.087	2337.442	428.000	4.280
10	9476.180	2324.941	437.000	4.370
15	9425.585	2358.445	403.000	5.210
20	9437.987	2354.943	396.000	5.280
30	9534.303	2377.449	371.000	6.720
40	9572.718	2354.949	360.000	7.900
50	9504.073	2339.950	360.000	8.790
75	9448.435	2332.452	454.000	13.290
100	9378.277	2324.951	482.000	15.430
125	9351.744	2316.457	547.000	20.890
150	9304.128	2279.960	618.000	23.700
200	9159.688	2259.959	753.000	30.000
250	9110.531	2249.964	806.000	36.740
300	8768.228	2177.465	1015.000	43.470
500	8517.211	2097.466	1390.000	69.499
750	7820.611	1919.974	1950.000	94.439
1000	7359.828	1814.976	2462.000	123.099
2500	4764.175	1189.989	4824.000	241.199
5000	3408.635	824.992	6120.000	305.998

minimized with 10 granules. With fewer granules, the request failure rate caused enough re-requests for locks that the overall CPU costs for locking increased. With more than 10 granules, the reduction in lock request failures did not offset the costs of setting the additional locks required for each transaction.

Table 2-8
Transaction throughput measurements
Initial Scenario

NO_of_GRANULES	AVERAGE RESPONSE TIME	COMPLETED
1	751.914	128
2	557.232	168
3	534.399	178
4	523.082	182
5	490.297	195
7	506.667	189
9	515.117	188
10	472.330	203
15	484.214	196
20	462.678	208
30	472.732	205
40	454.189	212
50	441.537	218
75	430.543	223
100	420.416	231
125	463.255	208
150	460.429	210
200	435.748	222
250	504.021	192
300	447.065	215
500	472.088	204
750	570.089	168
1000	546.023	175
2500	815.784	115
5000	1054.988	86

The average response time and the total number of transactions completed at time TMAX reached extremums at 100 granules. With this number of granules, the smaller transactions requiring less resources were able to run to completion. Thus, a 'shortest job first' property was observed. Moreover, with finer granularity (>200 granules) locking overhead actually increased the average response time. In these cases the higher I/O locking overhead (753 to over 6000 time units) delayed the normal processing of transactions.

In summary, under the initial scenario parameter settings, the useful computer utilization increased as the number of granules increased then leveled off and fell. Moreover, the maximum utilization occurred with a relatively small number of granules and that utilization was within 1% of that optimum for 10 granules. The conclusion can be drawn that crude locking schemes with coarse granularity were nearly optimal. Since a crude locking system may be easier to implement than a sophisticated finer granularity scheme, it may be preferred.

For this case, response time, and throughput were all better with a small number of granules. Hence, a large number of granules (such as would be required to lock disk sectors or individual records) may be inappropriate.

However, changes in the parameters and simulation model do alter these observations. In the next section, the effects of alternate workload parameters are reported. In section 3.3, the systems parameters are varied. In section 3.4, the effects of two extensions to the model are reported.

3.2. Effects of Workload Parameters

Changes in the workload parameters would reflect changes in the characteristics of the applications which were running on the system. It has been noted already that the number of transactions had little effect on the observed output parameters. Other workload parameters did make some difference on the optimum granularity. The major difference was due to the lock placement assumptions. Other workload parameters tested included changes in transaction sizes, changes in database sizes and the addition of an idle time period for the transactions.

3.2.1. Placement of Locks

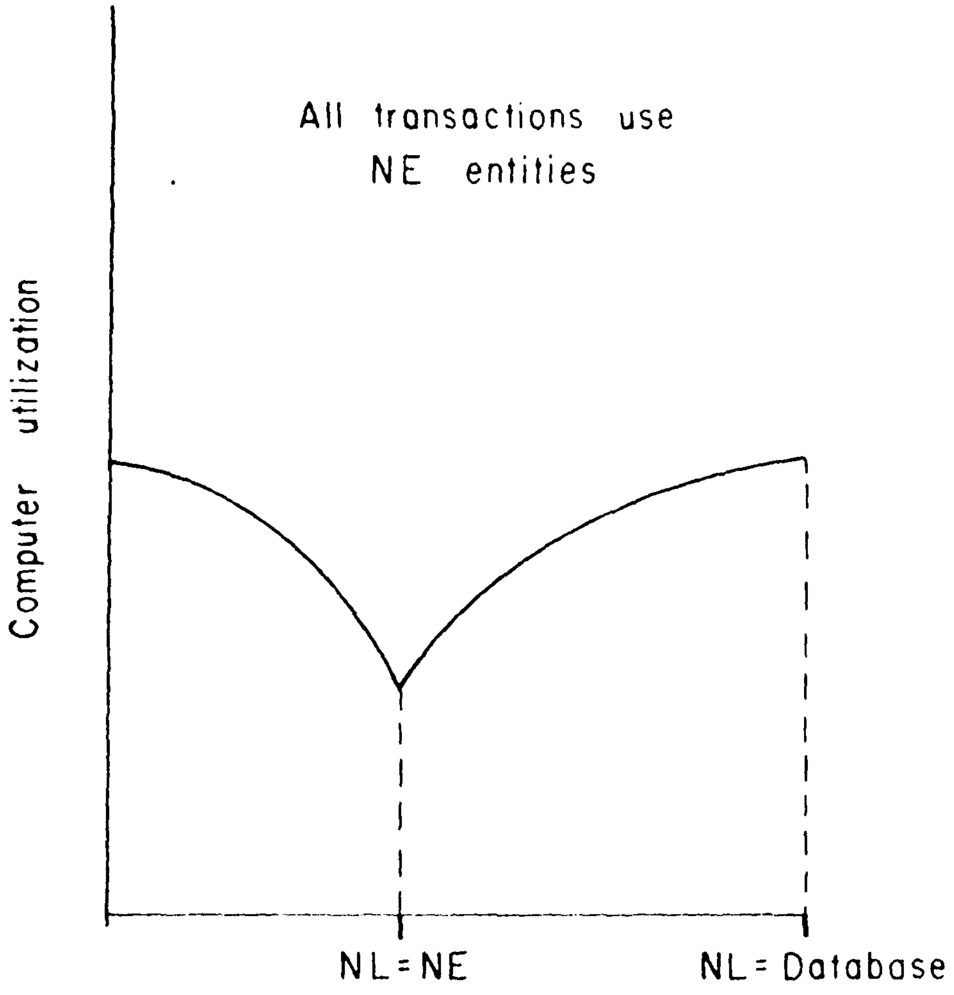
In the previous experiments the locks were assumed to be well-placed. The other two placement assumptions were also tested. In the worst case assumption, each transaction required the maximum number of granules possible. In the random placement assumption, the probability of accessing any entity was identical and independent of any previous entities accessed.

Which model is chosen affects the previous observations. If the "worst case" is chosen, the following intuitive analysis applies. In figure 2-3 it is assumed that all transactions touch the same number of entities, NE. The machine utilization measures would decrease as the number of locks for the entire database increased from one to NE. The decrease is because each transaction would require more locks thus increasing the locking overhead. However, there would be no additional parallelism because each transaction locked the entire database.

The utilization would increase, however, as the number of locks increased from NE to the total number of entities in the database. In this case, the cost of the locking overhead would remain constant while the allowed concurrency increased. The locking overhead would remain constant since each transaction could never set more than NE locks.

Consequently, the optimum number of locks would be very dependent on the transaction sizes in the worst case placement lock assumption. Moreover, it would always occur at 1 granule or the maximum number of granules (corresponding to one lock per entity) if all the transactions were the same size. The effects of having varying transaction sizes will be discussed below.

The simulation model was run for each of the three placement assumptions under a wide variety of parameter settings. Figures 2-4 and 2-5 diagram some of the



Number of Granules

Figure 2-3: Expected Computer Utilization under the Worst Case Lock Placement

results. In figure 2-4, the transaction sizes were determined by an exponential distribution with a mean value of 500 entities (10% of the database). In figure 2-5, the transaction sizes were also determined by an exponential distribution but with a mean value of 5 entities (0.1% of the database). For these runs, the locks were assumed to be in main memory (no lock I/O required) and the I/O and CPU time required by the transactions were equal. These conditions were chosen as the ones most favorable to finer granularity. The other parameters were identical to those described in the initial scenario, with one major exception. In figures 2-4 and 2-5, the random lock conflict assumption is assumed for all three placement conditions. Under the random lock conflict assumption, the granules associated with each transaction are considered to be completely uncorrelated. This modification is made primarily for validity checking. With the same lock conflict assumptions, the end points (1 and 5000 granules) should and did result in identical simulation runs under the three lock placement assumptions.

The top curves in both figure 2-4 and 2-5 were consistent with the results of the initial scenario. The bottom two curves represent the worst case and random access assumptions.

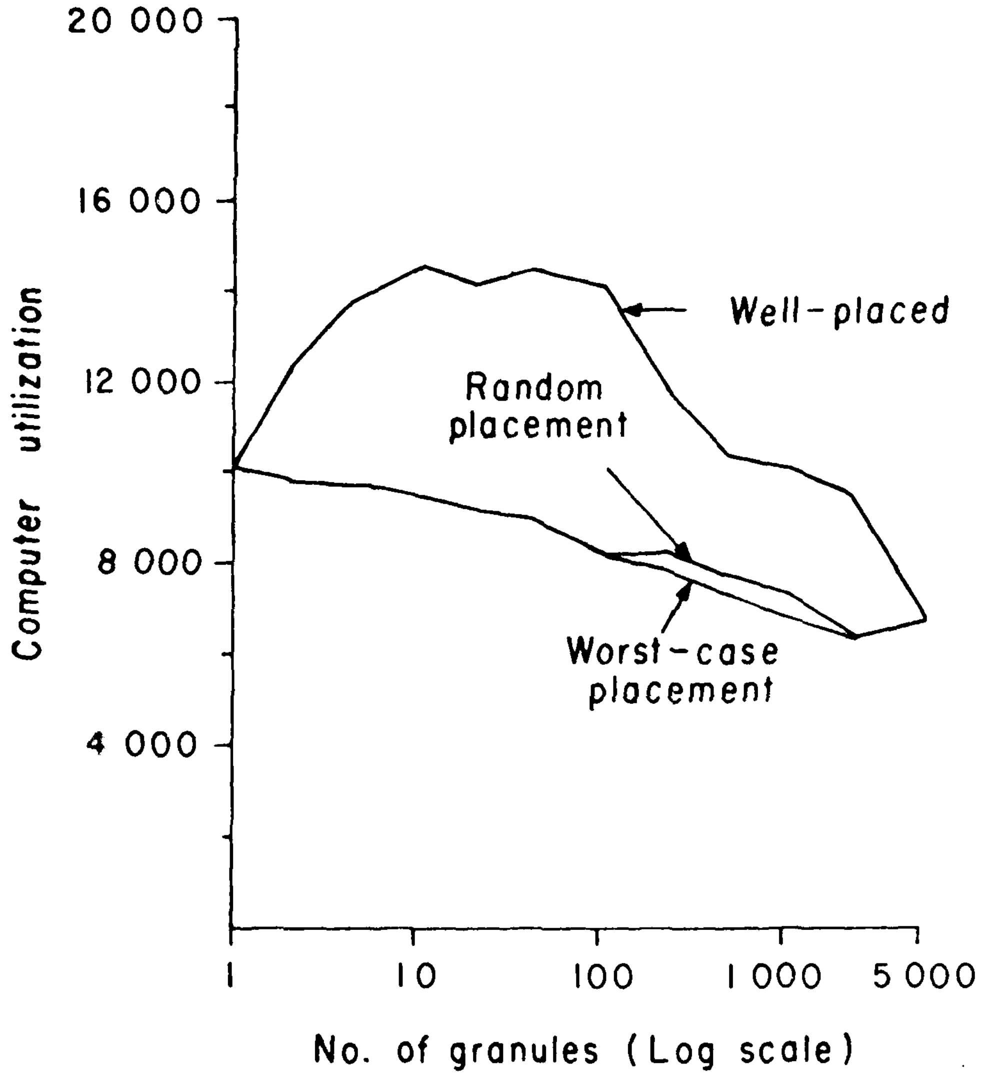


Figure 2-4: Computer utilization as a function of no. of granules.
Large transactions and Different Lock Placement Assumptions

For large transactions requiring about 10% of the database (see figure 2-4) a smaller number of granules was still to be prefered to a lock for each entity. For small transactions requiring about 0.1% of the database (see figure 2-5) one lock per entity produced the greatest machine utilization under the worst case and random placement assumptions. However, even with small transactions, the degree of improvement was small as the granularity increased beyond a certain limit. For example, 90% of the maximum machine utilization was reached with 200 locks.

Next, the simulation was run with mixed size transactions (AMEAN = 250, BMEAN = 5, ALPH = .1) using the best case, the worst case and random access assumptions. Intuitively, this simulates a few large transactions and many small ones. As previously stated, under the well-placed assumption a small number of granules is best. A relatively flat curve relating machine utilization and the number of locks is observed for the worst case and random access assumptions. Thus, in these two cases, the granularity of locks, whether coarse or fine, did not greatly effect the useful machine utilization. In fact, 98% of the maximum utilization was achieved with both 10 and 2500 granules. The basic problem with fine granularity was that the expense of running just a few large transactions seemed to outweigh the gain due to the increased con-

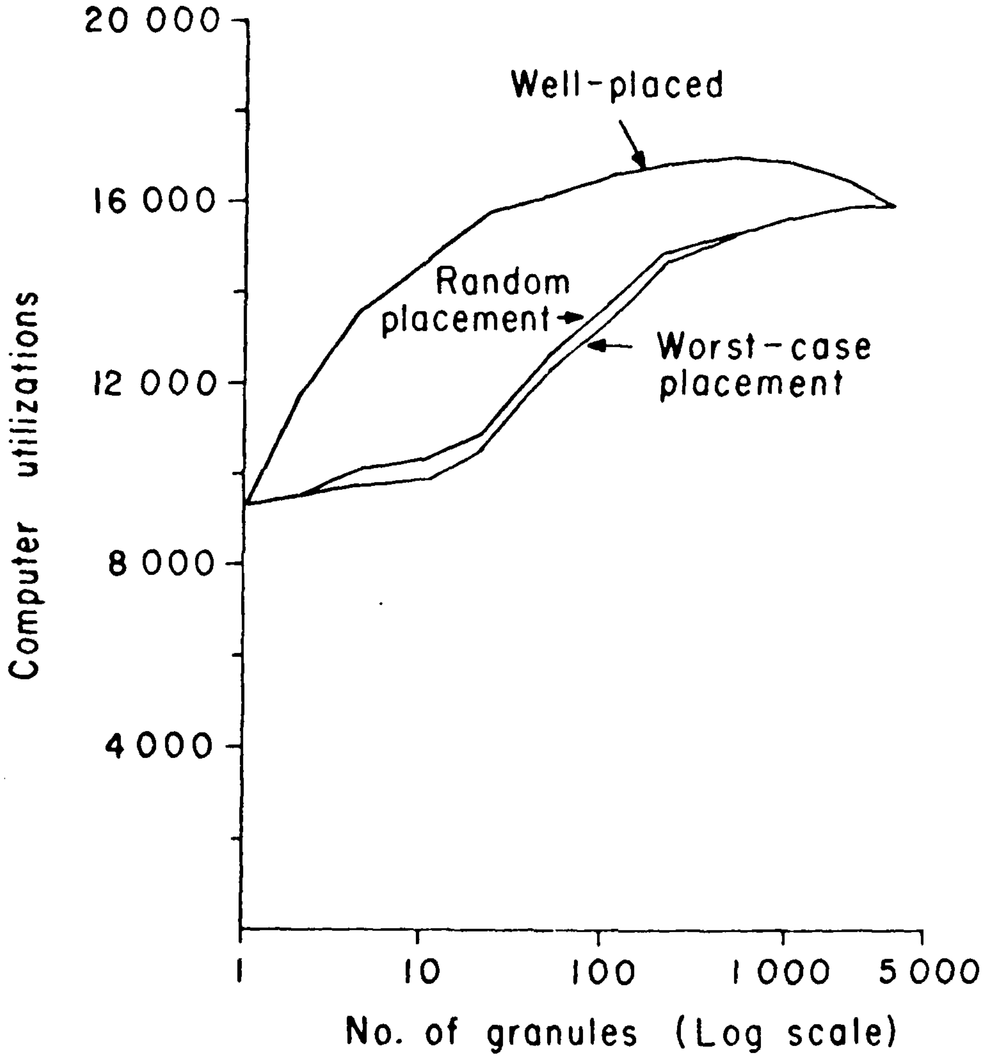


Figure 2-5: Computer Utilization as a function of no. of granules. Small Transaction and Different Lock Placement Assumptions currency experienced by the small transactions.

3.2.2. Transaction Size

Under a uniform distribution of transaction sizes, the number of entities required by a transaction was determined by the RAD parameter. The simulation was run under the well-placed lock assumptions with RAD values of 1, 25, 50, 100, 250 and 500 on a data base containing 5000 granules. The first case results in an initial average transaction size of 1/1000 th (1*NTRAN/2)of the database. The last case on the other hand, results in an average transaction size requiring one half (500*NTRAN/2) of the entities in the database.

As the needs of the transactions increased, maximum machine utilization and throughput were obtained with fewer and fewer granules. Minimum response time behaved similarly. The optimum 1% and 5% intervals of useful I/O are presented in figure 2-6. Note that even for very small transactions, 95% of the optimum was reached with as few as 10 granules.

The two other distributions of the transaction sizes were also tested in order to model different transaction environments. With an exponential distribution, with the

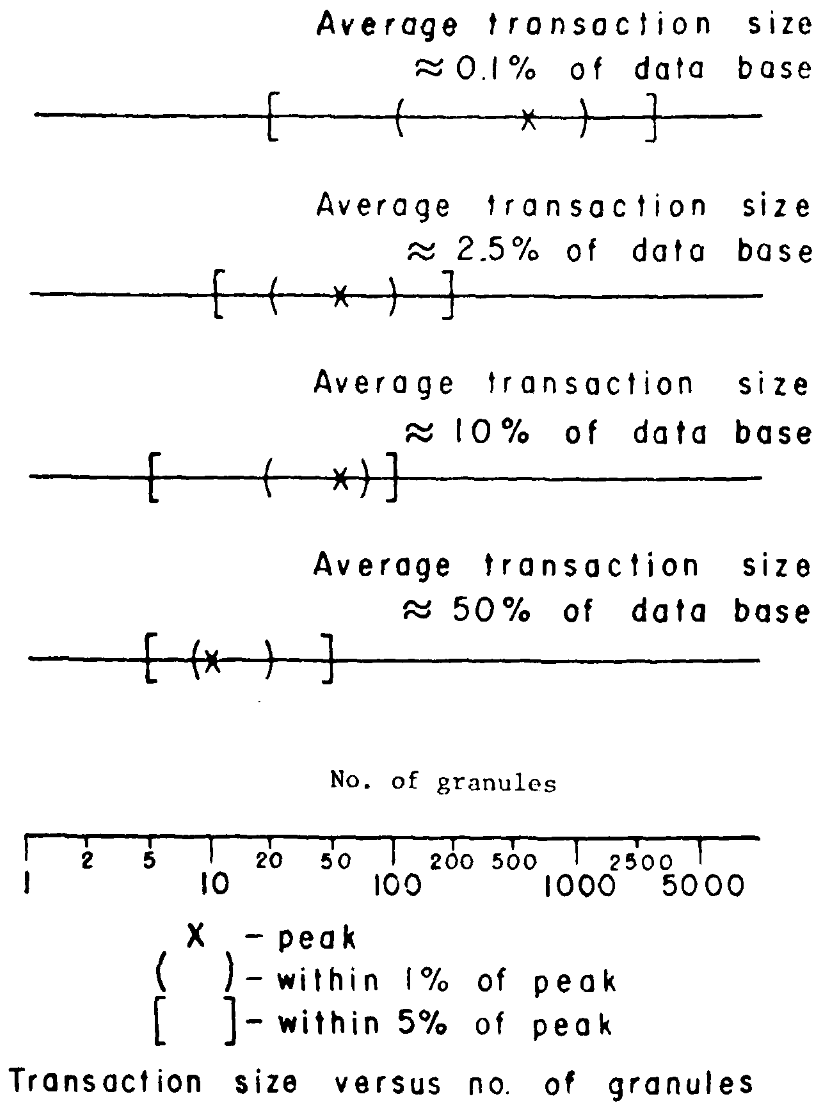

Average transaction size ≈ 0.1% of data base
Average transaction size ≈ 2.5% of data base
Average transaction size ≈ 10% of data base
Average transaction size ≈ 50% of data base
No. of granules
1
2
5
10
20
50
100
200
500
1000
2500
5000
X - peak
( ) - within 1% of peak
[ ] - within 5% of peak
Transaction size versus no. of granules

Figure 2-6

same mean value as the uniform distribution (AMEAN=RAD*NTRAN/2), the results were very similar. For a small AMEAN, say 5 entities, 500 granules were optimal. Again, however, 10 granules produced useful machine utilization within 5% of the utilization realized with the 500 granules. With an exponential distribution and an AMEAN value of 250 entities, on the other hand, 40 granules was again optimal. In that case, the larger transactions realized too much locking overhead with the less coarse granularity.

However, with a hyper-exponential distribution, the "large" transactions (those determined by the BMEAN parameter) dominated the processing. Thus coarse granularity was again favored. For example, with AMEAN, BMEAN, and ALPH values of 5, 250 and 0.1 respectively, an NGRAN of 50 still produced the maximum useful computer utilization. In this case, the average transaction size was about 30 entities. But 10 percent of the transactions accessed on the average 250 entities and these transactions dictated coarse granularity.

The simulation was also run under the random lock placement assumptions varying the granularity and transaction sizes. For these experiments, the IORATE and CPURATE were again equal and the LIORATE was set to zero. The other parameters were identical to those described in the initial scenario. With an average transaction size of less than 25 entities, the finest granularity was again optimal. When the average transaction size was between 25 and 50 entities, the useful computer utilizations at 1 and 5000 granules were approximately equal. With an average transaction size greater than 50 entities (1% of the database), one granule was optimal.

3.2.3. Database Size

Simulation experiments were also run with various granularities on a database of 50000 entities. The average transaction size was fixed at 250 entities and the simulation was run for 15000 time units. The effects of increasing the database size was similar to the effects of decreasing the transaction size. With well-placed locks, the optimal granularity occurred at 500 granules. In this case, five percent of the maximum utilization was realized with 20 to 2500 granules. With random lock placement, the finest granularity was again optimal.

3.2.4. Idle Time

For some applications, locks can be held while a user or application program pauses for some duration (often thought of as "head scratching"). The simulation was modified to reflect this effect by holding all locks for an idle period of 100 time units (say, for example, about 25 seconds in the interpretation mentioned at the beginning of this section). The simulation was then run with the parameter settings of the initial scenarios shown in Tables 2-5 and 2-6.

The results were remarkably similar to those in Table 2-2. The useful I/O curve had slightly more variation than the curve in figure 2-2 with a peak occuring at 50 granules. Ten granules still produced useful I/O and CPU times within 5% of the optimum. Hence a small number of granules was still best even with substantial pauses in the transaction processing.

3.2.5. Workload Parameter Summary

The lock placement assumptions clearly had the most dramatic impact on the machine utilization as a function of locking granularity. The second most important parameter was the size of the transactions accessing the database.

Fine granularity may be best if the following two conditions were meet: 1) almost all of the transactions are small and 2) access patterns are random with no sequentiality. Under these conditions, the greater the number of locks. the greater the machine utilization. However, the rate of increase dropped dramatically after a certain level of granularity was obtained (about 200 granules in our simulation). Hence "medium" granularity did almost as well as fine granularity; coarse granularity was unacceptable in these cases.

If too many of the transactions access a large portion of the database, fine granularity produces too much locking overhead and coarse granularity was again to be preferred.

Regardless of the transaction sizes, if the data access patterns were primarily sequential, coarse granularity was still the most effective.

3.3. Effects of the System Parameters

The locking granularity, determined by NGRAN, has been the major system parameter studied so far. This parameter is clearly the one over which the system implementors have the most control. The effects of the other system parameters on the system throughput and utilization are presented below. In particular, the IORATE and IOOVLP parameters were varied in order to "balance" the I/O and CPU requirements of the transactions. Also, the LIORATE and LCPURATE parameters were varied to control the locking overhead. In addition, for each parameter, its interaction with the locking granularity is also discussed.

3.3.1. I/O versus CPU Balance

The effects of the ratio of the required I/O time to the required CPU time per entity was investigated. The CPU rate (CPURATE) per entity for a transaction was held fixed at .05 units/entity. The simulation was run with I/O rates (IORATE) per entity set at .01, .05, .1, .2, and .3. For each setting of the I/O rate, the number of granules (NGRAN) was varied from 1 to 5000. The lock I/O rate per granule was set equal to the I/O rate per entity in order to reflect the locks being on the same speed device as the data. Each simulation ran for 5000 time units. The other input parameters had the values indicated in Tables 2-5 and 2-6.

Under the well-placed lock assumption, the useful I/O curves for each setting of IORATE were bell shaped and heavily skewed towards a small number of granules. As such they were similar to the curves in figure 2-2 and are not repeated here. The peak of these curves occurred with somewhat finer granularity as the IORATE came closer to the CPURATE. With a system balanced between I/O and CPU requirements the maximum utilization of both CPU and I/O resources was possible. However, even with balanced transactions, 100 granules were sufficient to achieve the maximum machine utilization. With CPU bound transactions (CPURATE >IORATE) within 5% of the peaks was reached with as few as 10 granules. Varying the IORATE had little effect on the throughput measurements (average response time, and number of transactions completed) as a function of the number of granules allowed. The useful CPU time, as a function of granule size, showed a similar distribution as the useful I/O. The costs associated with locking were again minimized with 100 granules.

Under the random and worst case placement assumptions and small transactions, the finest granularity was optimal regardless of the I/O to CPU balance.

3.3.2. Multiple I/O Paths

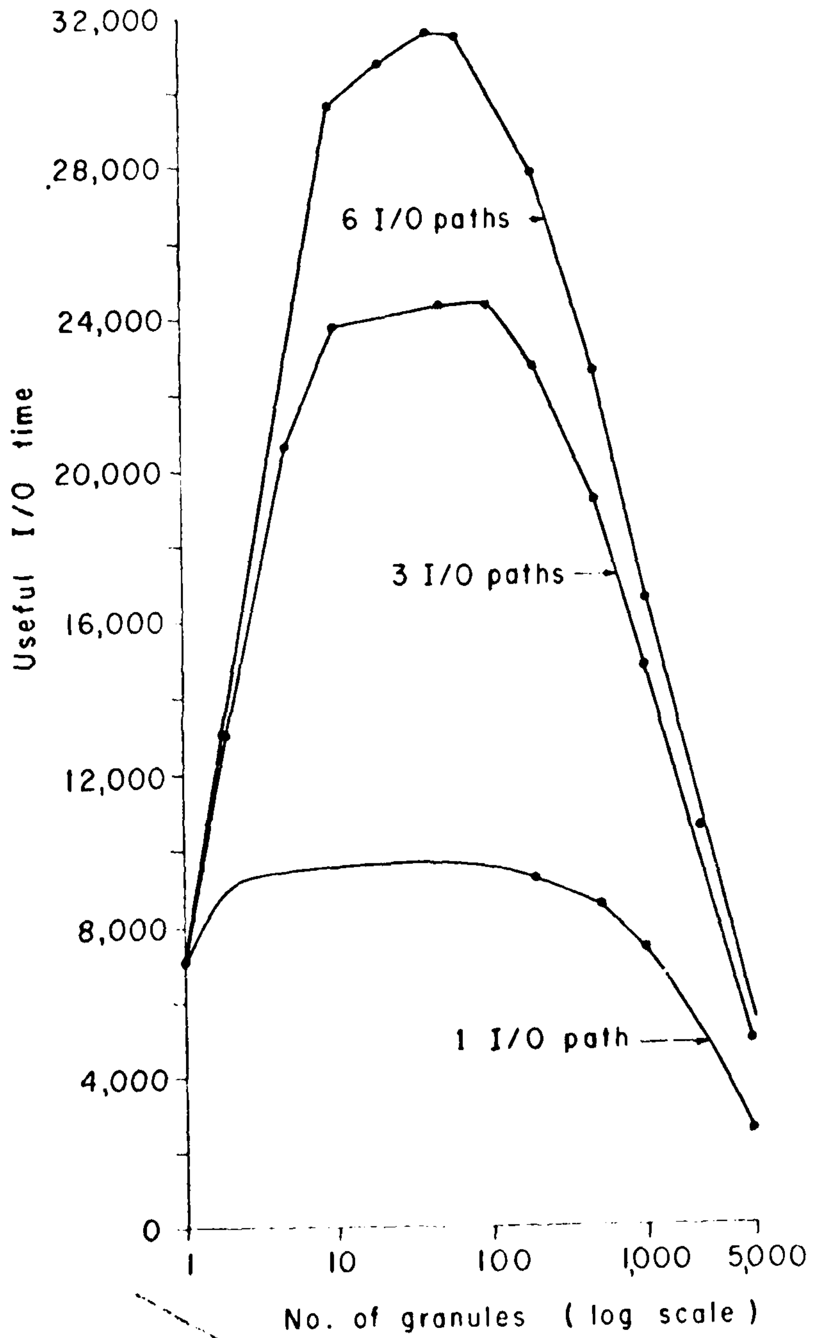
One method of "balancing" a system that is I/O bound is to increase the number of I/O channels to main memory. In the previous runs, the IOOVLP value was one. These experiments thus simulated a system with one I/O path between main memory and secondary storage. In the next series of runs, this parameter was set to three and six to simulate, for example, a database environment with three and six disk drives respectively. Other input parameters were the same as in Tables 2-5 and 2-6.

Except for greatly increased magnitude, the output parameters had a similar distribution as those in Table 2-7. The useful I/O time (USEFULIO) versus the granularity, for simulation runs under the well-placed lock assumptions, are shown in figure 2-7. Note, with 10 to 100 granules, the useful I/O increased by a factor of about 2.5 for three I/O paths as compared to the useful I/O with one I/O path. (The best results possible would be increased useful I/O by a factor of 3.) Moreover, as the number of granules increased three drives became less and less effective. For 2500 granules, for example, only a 1.5 factor increase in useful I/O was realized. The results for six I/O paths were similar. Ten to one hundred granules tripled the increase in useful I/O. With 2500 granules, the increase in useful I/O was slightly less than doubled.

In the random and worst case lock placement experiments, the finest granularity was again favored as additional parallelism was made possible.

3.3.3. Lock I/O Costs

In the previous experiments, the lock I/O rate (LIORATE) was equal to the transaction I/O rate (IORATE).



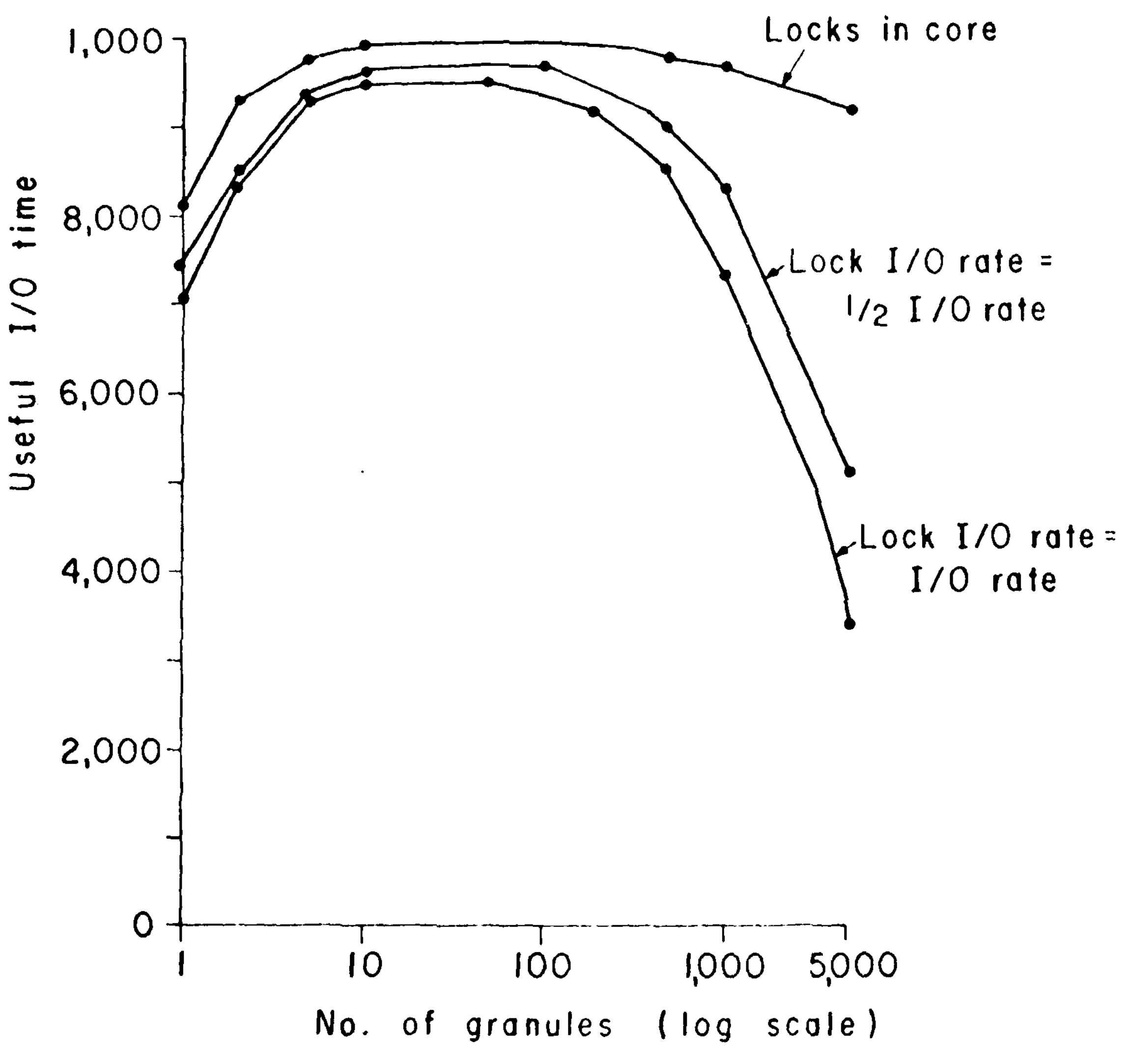
EFFECTS OF MULTIPLE I/O PATHS
Well-placed lock assumption.
Figure 2-7

In the next series of simulation runs, only the lock I/O rate and the granularity were varied. The simulation was run with other parameters as in Tables 2-5 and 2-6. The useful I/O times (USEFULIO) for the well-placed lock assumptions are shown in figure 2-8.

As the lock I/O rate decreased, a larger number of granules could be afforded before the advantages of more parallelism were outweighed by the locking overhead. Of particular interest is the situation where the LIORATE was zero. This case is analogous to keeping all locks in main memory. Even with no lock I/O costs, there was a very flat extremum for USEFULIO between 10 and 200 granules. Having a granule correspond to fewer than 25 database entities (number of granules > 200) resulted in noticeably poorer performance. If the interpretation of an entity is a 512 byte page (or a 4096 byte sector) a database management system should thus not 'protect' less than 13,000 (or 100,000) bytes of data with one lock.

3.3.4. Lock CPU Costs

The CPU costs for setting one lock were dependent on the lock management algorithms. To investigate the effects of varying the CPU rate for locking on the desired granularity, the simulation was run with CPU lock



EFFECTS OF LOCK I/O RATE

Productive I/O Utilization versus no. of granules

FIGURE 2-8

(LCPURATE) costs per lock of .005, .01, .025, .05 . .075, and .1. For this series of experiments, the LIORATE was set to zero to simulate the effects of maintaining all locks in main memory. Other parameters were as in Tables 2-5 and 2-6.

Under the well-placed lock assumption and a small number of granules, the CPU lock costs (LOCKCPU) were approximately linearly proportional to the CPU rate per lock (LCPURATE). In these cases, there were enough unused CPU resources available for locking. For a large number of granules, however, the CPU lock costs increased slightly less than linearly with LCPURATE. In these cases, the locking CPU utilization interfered with normal transaction processing. For all CPU lock costs tested, however, the minimum locking costs occurred at 10 granules.

Under the well-placed lock assumption the maximum amount of useful CPU and I/O occurred with 10 to 100 granules and was about the same regardless of the lock CPU rate. With lock CPU rates of less than 1 millisecond (LCPURATE = .005), the peak occurred at 100 granules; within 1% of that peak occurred with 10 to 1000 granules. With lock CPU rates between 1 and 5 milliseconds (LCPURATE = .005 to .03) the peak was at 50; but the useful I/O and

CPU dropped off sharply with more than 200 granules. With higher lock CPU rates, 10 granules were optimal and at most 100 granules for the locking granularity were affordable.

Simulation experiments were also run varying the lock CPU costs under the random lock placement assumption. In these experiments all transactions were small (AMEAN = 5, ALPH = 0) and the LIORATE was again set to zero. The CPU and I/O rates for transaction processing were both about 30 milliseconds per entity (IORATE = CPURATE = .2). In these experiments, an increase in lock CPU rates greatly affected the computer utilization at the finest granularities. With a 5 millisecond lock cost (LCPURATE = .03), the useful computer utilization was 5% of the utilization observed with a 2.5 millisecond lock overhead cost (LCPURATE = .015). However, the finest granularity was still optimal until a 15 millisecond per lock overhead cost (LCPURATE = .1) was incurred.

3.3.5. System Parameter Summary

Some of the system parameters did suggest somewhat finer granularity under the well-placed lock assumptions. In particular, two factors had some effect on the optimum granularity. When the resources expended for locking were reduced, finer granularity was affordable. With lock I/O costs of zero and the lowest setting of lock CPU costs, 100 locks was optimal. Even in these cases, though, too fine a lock granularity was not acceptable.

The second factor which had an effect on the optimum granularity was the balance between the CPU and I/O resource needs of the transactions. Under a balanced system load and the well-placed lock assumption 50 to 100 locks were again optimal.

Under the random and worst case lock placement assumptions, in most cases, the lock cost parameters (LCPURATE, LIORATE) did not change the optimal granularity. The other system parameters had no affect on the optimum granularity under these placement assumptions.

3.4. System Extensions

In the previous experiments all granules were assumed to be the same size and all of the locks were acquired at the beginning of a transaction. In this section two modifications to the model are introduced to study alternate assumptions. In the first extension a lock hierarchy is simulated. In a lock hierarchy, transactions of different sizes lock different sized granules. In the second extension, a "claim as needed" locking strategy is simulated. In that strategy, transactions acquire locks as they need the corresponding entities.

3.4.1. Lock Hierarchy

In many of the previous experiments it is noted that the expense of locking a large number of granules by a large transaction offsets any increase in parallelism realized by fine granularity. One way a large transaction can avoid the expense of locking many small granules might be to have the large transactions lock large granules while the small transactions continue to use the small locks [GRAY76].

3.4.1.1. The Model Extension

In the simulation extension a two level hierarchy was implemented. A transaction, depending on its size, either requested a set of small locks or one global lock which covered the entire database.

With this extension, we explored the interactions between any two levels of a more general hierarchy. A more general hierarchy could be any tree-like graph structure. A transaction could lock the root of a subtree and thus control access to the parts of the database covered by any locks in that subtree. Alternately, the transactions could mark the root of the subtree to indicate that locking is taking place at a lower level. The transaction would then treat each offspring of the root as its own subtree.

In the extended model, the choice between the global locks and the small locks simulates the choice between the root of one subtree and its direct descendents. The performance tradeoffs between increased parallelism and increased locking overhead of a more general hierarchy occur similarly at each node. Thus, the results of this extension can be applied to the more general hierarchy and a more complex tree structure need not be simulated.

The simulation was modified by adding 'pending' and 'blocked' queues for the global lock. If a transaction was "small", it set the global lock in shared mode and was placed on the original pending queue. From that queue the "small" transactions competed for the small locks as in the original model. If the transaction was "large", the global lock was set for exclusive access and the transaction waits for all active transactions to finish. With the global lock set for exclusive use, new transactions, regardless of size would also wait in the blocked queue. Once the large transaction was allowed to proceed, it went directly to the I/O queue bypassing the small lock control.

The simulation was used to study the effects of certain parameters of such a hierarchy on the desired granularity. One of the main areas of interest was the criteria for deciding whether the small locks or the global lock should be used. An input parameter was added to the simulation which specified the threshold percentage, TP, of the database which must be touched by a transaction before it was declared "large". If a transaction used less than TP percent of the database, the small locks would be used. Otherwise, only the global lock would be set.

3.4.1.2. The Simulation Results

The simulation was run with threshold percentages of 0.1%, 0.2%, 0.5%, 1%, 2%, 5%, 25%, 50% and 100% for each of a large number of other parameter settings in order to find the value of TP which maximized useful machine utilization. The optimum threshold observed was dependent on the number of small locks, the assumptions concerning the placement of those locks, the number of entities touched by the transactions, and the size of the database.

Figure 2-9 shows the effects of the threshold percentages on machine utilization in two instances with different numbers of small locks. For both of those cases [illegible] sized transactions were used and well-placed locks

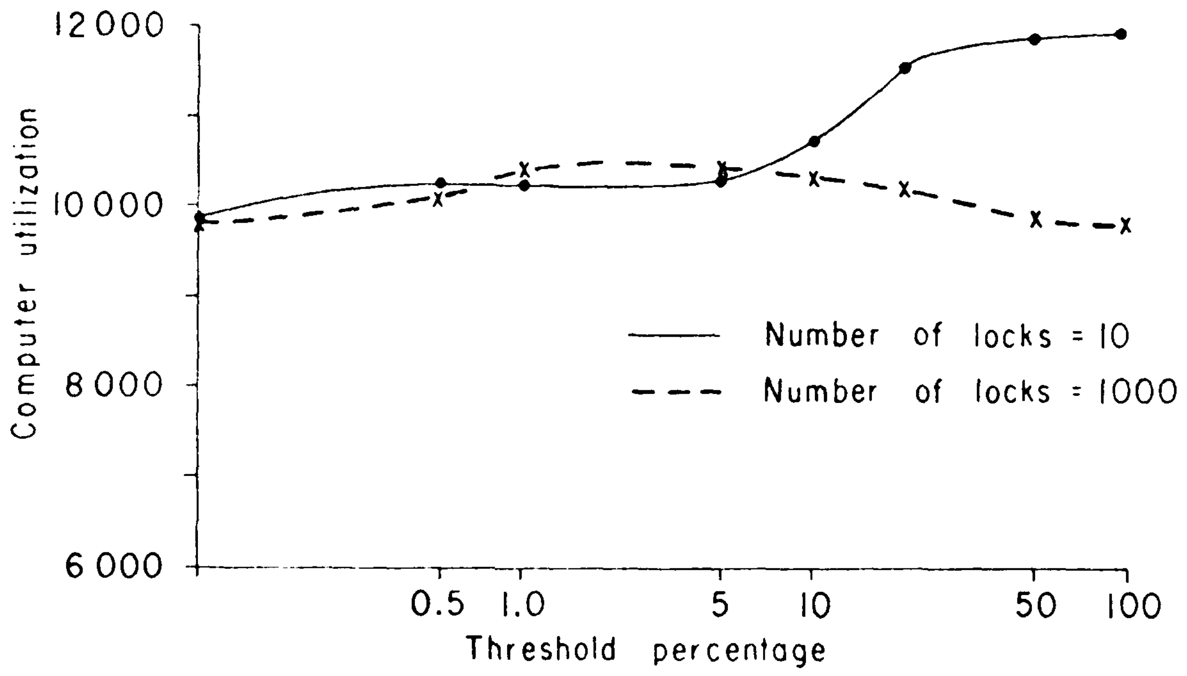


Figure 2-9: Computer Utilization versus Threshold Percentage in a Lock Hierarchy.

were assumed. With ten small locks, the maximum machine utilization was reached with thresholds of 50 and 100 percent. A threshold of 100% resulted in all transactions using the small locks, i.e. as if no hierarchy were present. However, with 1000 small locks, for example, a threshold of 5 percent was optimal. Changing the assumption about the placement of the locks also made a dramatic difference.

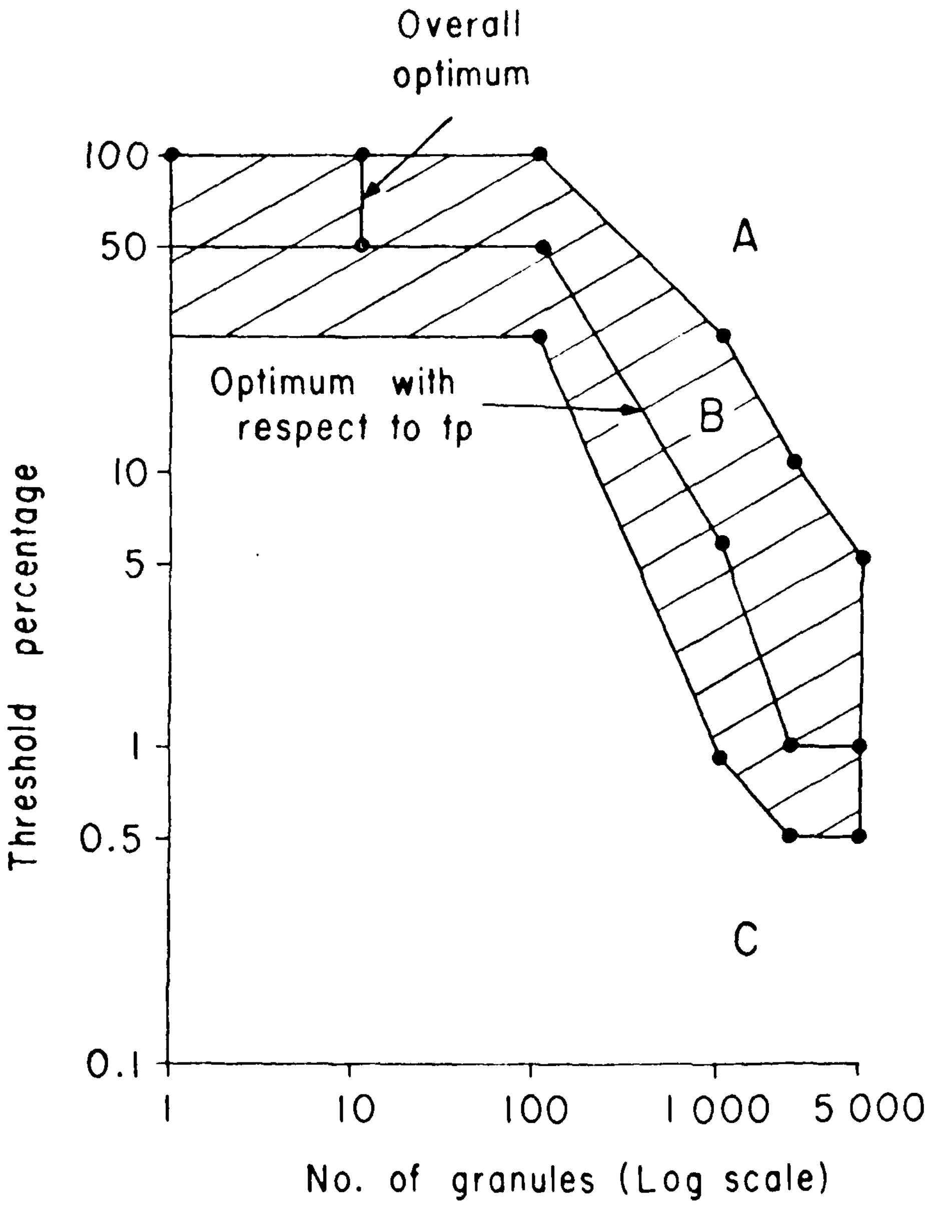
Figures 2-10 and 2-11 further explore the effects of the number of small locks on the threshold percentages. The results in Figure 2-10 reflect the "well-placed" assumption. Random access to the database was assumed for the simulation results shown in figure 2-11.

Each of the graphs is divided into three areas based on machine utilization. The "optimum" line represents the threshold value, TP, at which the maximum I/O and CPU utilization was observed for a given number of small locks. With threshold values in area B, the hierarchical locking produced results within 2% of that maximum utilization. In area A, the utilization was less than in area B. In this case, too few transactions used the global lock, i.e. the threshold, TP, was too high. In area C, the machine utilization was also less than in area B. In this case, however, too many transactions used the global lock, i.e. the threshold, TP, was too low.

For example, consider figure 2-10 with 1000 small locks. The machine utilization increased as the threshold percentage was increased from 0.1% to 5%, but decreased as the threshold increased from 5% to 100%. However, simulation runs with threshold percentages between 1% and 25% produced within 2% of the machine utilization observed with the optimum threshold.

In figure 2-10, "well-placed" granules were assumed. With more than 1000 small locks the optimum value of TP was between 1% and 5%. With the number of locks between 10 and 100, TP values of 50% to 100% were optimal. In this granularity interval, the 2% area included the case where all transactions used only the small locks. The overall maximum machine utilization occurred in figure 2-10 with 10 locks and TP values greater than 50%. In these cases, almost all of the transactions used the small locks. Hence, the value of a lock hierarchy under the well-placed locks assumption was very small.

However, in figure 2-11, random lock placement was assumed. With coarse granularity, the optimum threshold occured at 0.5%. With a higher threshold, more of the smaller transactions would use the small locks, and consequently would lock a large portion of the database. As a result, these transactions would expend more resources for locking than if the global lock were used without signifi-



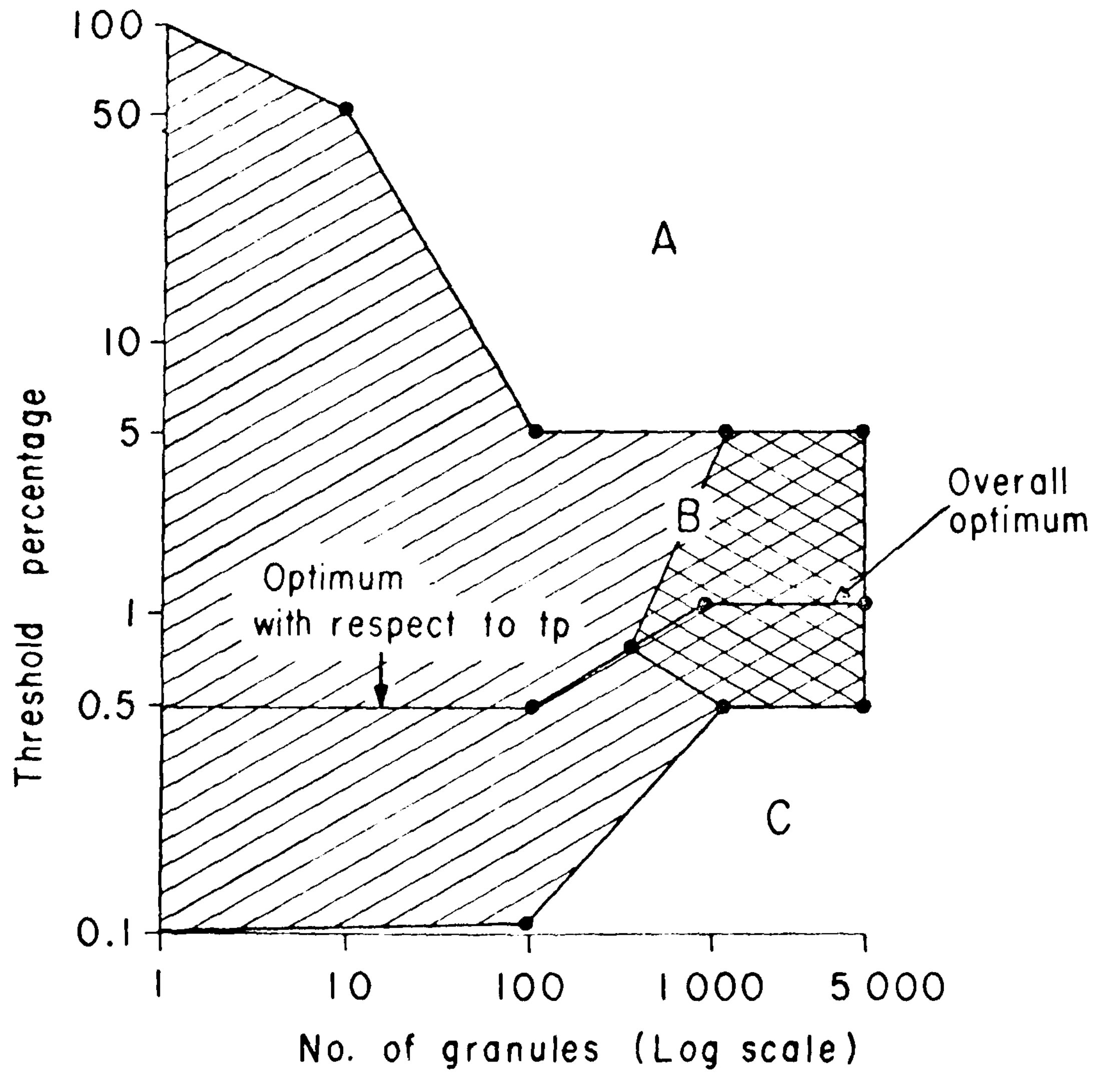
Area A: tp too high
Area B: Highest Machine Utilization
Area C: tp too low

Figure 2-10 The Effects on Computer Utilization of the Number of Granules on the Optimal Threshold Percentage with Well-placed Locks in a Lock Hierarchy

cantly increasing the concurrency allowed.

In figure 2-11, the differences in computer utilization between areas A, B, and C was small for coarse granularity. For 10 granules, for example, no matter what value of TP was used, the computer utilization was within 3% of the maximum observed for that granularity. Similarly, for 100 granules, the computer utilization was within 15% of the maximum observed for any value of TP. Thus even with random lock placement, a hierarchy with a small number of small locks, at best, provided slight improvement over a single level locking system.

Under the random access assumptions, the overall maximum machine utilization occurred with 5000 granules and a TP of 1%. The cross-hatched area in figure 2-11 represents those combinations of TP and number of small locks which resulted in machine utilization within 2% of the overall maximum. Hence, fine granularity was preferred. The lock hierarchy effectively prevented excessive locking overhead for large transactions. The coarse granularity, on the other hand, resulted in poorer useful machine utilization regardless of the TP setting. With 10 granules, for example, the USEFULIO was only 3/4 of the maximum USEFULIO observed with 5000 granules and a TP of 1%.



Area A:	tp too high
Area B:	Highest Machine Utilization
Area C:	tp too low

Figure 2-11 The Effects on Computer Utilization of the Number of Granules on the Optimal Threshold Percentage

For fine granularity, the B areas in figures 2-10 and 2-11 had considerable overlap. For example, in figure 2-10, with 2500 small locks, the 2% of optimum interval occured with a TP between 0.5% and 10%. In figure 2-11, with the same number of small locks, the interval occurred with TP values between 0.5% and 5%. Thus, at this granularity, a TP between 0.5% and 5% could safely be chosen regardless of the randomness of the data access patterns.

In other simulation runs, as the average transaction size decreased, the range of acceptable TP values (area B) also decreased. With fine granularity, regardless of the transaction sizes, a threshold between 1% and 2% always produced machine utilization within 2% of the maximum.

With coarse granularity, however, changes in the size of the transactions, created non-overlapped intervals of acceptable TP values. In other words, no one value of TP could be chosen that would be correct for vastly different sized transactions. Thus much greater care must be applied to a hierarchy with coarse granularity. Furthermore, a stable transaction size environment must be assumed.

The size of the database was also varied. For example, the simulation was run with a database consisting of only 16 entities. In this scenario, the possible interaction of a page/record hierarchy was examined. An entity corresponded to one of 16 records in a page. The simulation was then used to model the effects of locking the whole page by the global lock, or locking individual records by the small locks. Some increase in machine utilization was observed with a threshold of 50%; but the increase over using no hierarchy at all was less than 4%. Again it appeared that a lock hierarchy covering only a small number of smaller locks was not worth implementing.

The simulation was also run with databases of up to 100,000 entities. The results were similar to the results produced with a database of 5,000 entities. For example, experiments were run where the average transaction size of most of the transactions was just 0.05% of a 100,000 entity database and the average size of a few large transactions is 1% of the database. In these cases, with the finest granularity (100,000 small locks), a threshold of 1% was still optimal.

Other simulation experiments used the worst case data access assumption and produced results very similar to those in figure 2-11.

3.4.2. Claim As Needed Locking

There is another difference between the original model and some database concurrency control implementations. In the original model, a "preclaim" strategy was assumed where all of the locks were acquired before any transaction processing took place. In some database systems, a lock is not acquired until the related entities were actually needed by a transaction. These "claim as needed" schemes are used either to reduce the total time locks are held and/or because the locks to be acquired depend on data values of entities already accessed. In these cases, some locks may have to be held while other locks were requested, and deadlock can occur [COFF71]. In this section the effects of a claim as needed scheme are examined.

3.4.2.1. The Model Extension

The simulation was modified by cycling each transaction through the I/O and CPU queues (see figure 2-1) once for each lock required. The total I/O and CPU times required for a transaction were the same as in the original model and were equally distributed among each of a transaction's cycles.

Between each cycle, a transaction requested one lock. If the lock was granted, the transaction went on the active queues. When a transaction completed its last cycle on the active queues, all its locks were released as in the original model.

If the lock was denied, the requesting transaction was placed on the blocked queue. The lock could be denied due to locks held by either another active transaction, or by a blocked transaction. In the latter case, the blocking transaction was on the blocked queue, and a deadlock condition could exist. If deadlock occurred, a victim was picked for backout. The locks held by the victim were released, any blocked transactions were freed, and any time spent on the active queues by the victim was added to a "lost time" total.

3.4.2.2. The Simulation Results

The modified simulation was run varying the sizes of the transactions, changing the lock placement assumptions, and with and without a lock hierarchy. Again it was assumed that there was no I/O cost associated with locking and that the transactions required equal amounts of CPU and I/O resources.

The results of these simulation runs were very similar to the results from the preclaim strategy. In all cases, a claim as needed strategy did not change the granularity required for maximum machine utilization.

For example, figure 2-12, shows the results of running the simulation with no hierarchy, well placed

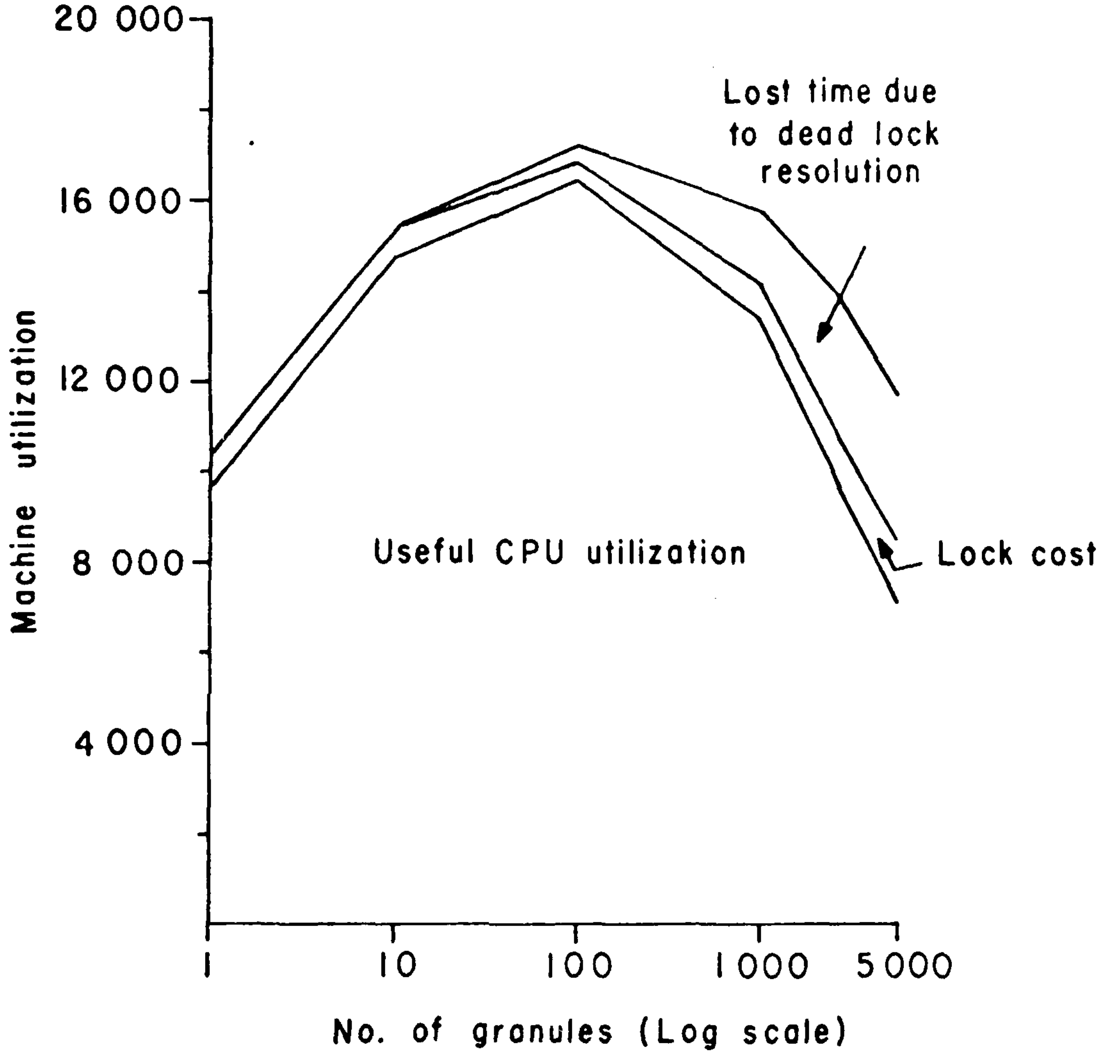


Figure 2-12 Computer Utilization as a function of No. of Granules with "Claim as Needed" Locking and Well-placed Locks.

granules and transaction sizes determined by a hyperexponential distribution. The lost time area included the machine utilization by transactions that had to be restarted due to deadlock. The useful computing included only the CPU resources used by successfully completed transactions.

In the simulation experiments, the locking cost observed in the preclaim model was greater than the locking cost observed in the claim as needed locking model. In the preclaim model, in the case of a lock request failure, all of the locks had to be requested again. In the claim as needed model, in the case of a lock request failure only the denied lock had to be rerequested. However, any decrease in lock costs in the claim as needed model was more than offset by the lost time due to restarting transactions. Thus, the useful machine utilization was greater under the preclaim model than under the claim as needed strategy. Many other cases with different transaction sizes and lock placement assumptions were also tested and produced similar results.

For example, figure 2-13 compares the useful machine utilization between the two models under the assumptions that all transactions were small and that each transaction had random data access patterns. In both of these runs, the average transaction size was 0.1% of the database.

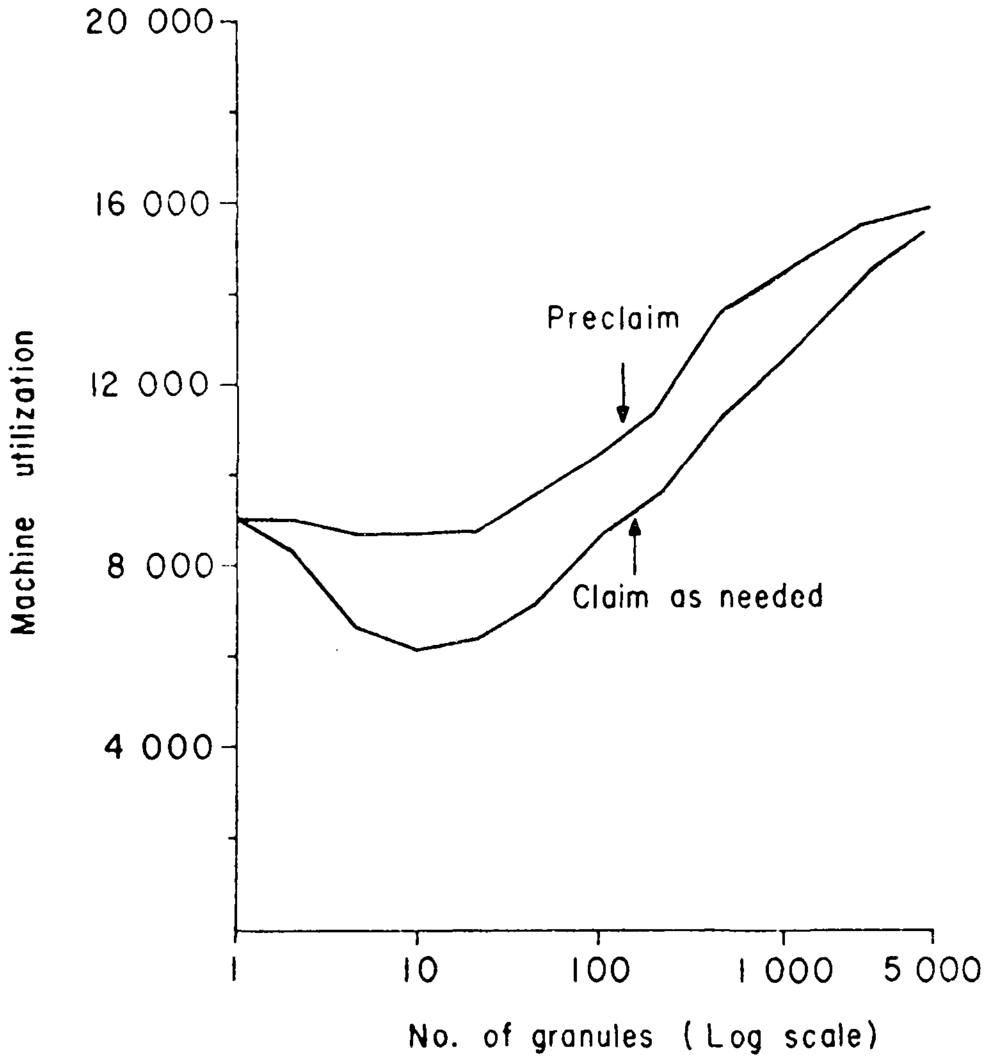


Figure 2-13 "Preclaim" versus "Claim as Needed" with Random Placement of Locks

AD-A103 141 CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB F/G 9/2
THE EFFECTS OF CONCURRENCY CONTROL ON DATABASE MANAGEMENT SYSTE--ETC(U)
APR 70 D R RIES N00039-78-C-0013
UNCLASSIFIED UCB/ERL-M79/20 NL

2 OF 3
AD A
103141

Note that, with the possibility of deadlock, the machine utilization curve did not flatten out as the granularity increased. Thus, the finest granularity is slightly more beneficial with the claim as needed model than with a pre-claim model. Note, however, that the claim as needed scheme again produced less useful I/O and CPU utilization than the preclaim model.

However, as the average transaction size became even smaller, the last observation did not hold. With an average transaction size of less than 0.05% of the database, random data access patterns, and the finest granularity, the claim as needed scheme resulted in greater useful machine utilization. Under these conditions, the claim as needed strategy allowed the greatest concurrency since locks were held for a shorter period of time. In contrast to other runs, very few transactions had to be backed out and the cost of rerunning such small transactions was insignificant.

The modified simulation was also run with a lock hierarchy and various threshold percentage values. A similarity in the shapes of the curves between the pre-claim and claim as needed strategies was also observed. Under the random access assumptions, for example, the maximum machine utilization was again reached with the finest granularity and a threshold value of 1 to 2 percent.

3.4.3. Summary

A locking hierarchy should be implemented when the small locks are of a fine granularity; a low threshold was used to separate the large and small transactions; and random data access patterns were anticipated. Under these assumptions the increase in machine utilization over a single level locking scheme was significant. Furthermore, a threshold of about one percent produced the best results independent of the granule placement or transaction size assumptions.

With coarse granularity, on the other hand, a locking hierarchy was not beneficial. The benefits of such a hierarchy were not significant and were only realized in certain cases. Another problem with the coarse granularity/locking hierarchy model was that the optimum value for the threshold percentage was extremely sensitive to the placement of the locks with respect to the transactions.

The acquisition of locks throughout the processing of a transaction did not significantly change the other conclusions. However, several observations were made. Deadlock detection and resolution appeared to be generally more expensive than the release and rerequest used in the preclaim strategy. Thus, when locks were known at the start of a transaction a preclaim algorithm is suggested.

4. CONCLUSIONS

The activity and effects of a locking mechanism were simulated to study the tradeoffs between increased parallelism of concurrently running transactions and increased overhead caused by sophisticated and complex locking mechanisms. The conclusions of the study are first applied to physical granules. The application of these results to predicate locking is then discussed.

4.1. Physical Locking

Under the assumptions mentioned in the description of the model, in many cases a small number of granules is sufficient to allow enough parallelism for efficient machine utilization. Furthermore, a large number of granules, corresponding to locking a page or record is often extremely costly.

These basic conclusions are due to the following observations. For large transactions, fine granularity becomes too expensive. A transaction which accesses half of the database, for example, would spend considerable resources locking each page. Yet little gain in parallelism would be realized since other transactions would have a strong probability of conflicting with the large transaction. A small transaction which accesses only one page, on the other hand, must lock a much larger granule. The resultant loss in parallelism is minimized because the small transaction would only hold the lock for a short period of time. The probability of conflict and the length of any waiting period would not be large due to that short period of time that the lock is held.

However, there are conditions where these observations do not hold. Details of which conditions support which level of granularity are presented below.

If equal sized lockable granules are assumed, a small number of granules (10 to 100) are sufficient under any of the following conditions:

1) The locks are well placed with respect to the running transactions.

2) The number of entities required by transactions vary in size and include at least some transactions that require access to a large number of entities.

3) Some portion of the locking scheme involves extra I/O for locking proportional to the number of locks.

However, each of the following factors supports somewhat finer granularity:

1) All of the transactions are extremely small and access less than 1% of the database.

2) The length of time that locks are held is extremely long and not proportional to the size of the transaction, as was the case with the "idle time" experiments.

3) The locking costs are reduced, for example, by keeping all locks in core.

4) A balance exists between the I/O resources and CPU resources required for processing a transaction.

If all of the following conditions are met, the finest granularity should be used:

1) All of the transactions are small.

2) The locking costs are reduced by, for example, keeping all locks in core.

3) Random access patterns (or worse) are exhibited by the transactions.

However, if condition 1 is violated, a lock hierarchy must be used if the fine granularity is still to be supported.

The overall conclusion is thus that the optimum locking granularity is somewhat application dependent. In many cases, coarse granularity, such as file or relation locking, with a preclaim strategy is to be preferred. In other cases, somewhat finer granularity, such as area or extent locking is best. In still other cases, the finest granularity such as page or record locking is required.

4.2. Predicate Locking

Four results from the simulation support the potential viability of predicate locking. Firstly, with predicate locking only a small number of locks must be maintained and can probably be maintained in main memory. The number of locks is proportional to the number of active transactions and not to the size of the database.

Secondly, while predicate locking may require more CPU time per granule than physical locking, the simulation results indicate that, for coarse granularity, some increases in locking overhead are affordable and do not significantly interfere with transaction processing.

Thirdly, the parameter which had considerable effect on the desired number of granules was the number of entities 'touched' by the transactions. As the transaction size decreased, the desired number of granules increased. Note, in predicate locking schemes, the portion of the database locked is determined by the transaction, and not a prespecified granularity, effectively mimicking the above variable granularity.

Finally, the results of the lock hierarchy simulation might indicate that a simple predicate locking scheme might be sufficient. In one such scheme, two types of locks could be supported. First an entire relation, record type or file could be locked. The small locks would be based on a simple unique key-value pair. The predicate locking scheme could easily check whether a key-value pair conflicted with either an entire relation lock or other key-value pairs. The lock hierarchy simulation results indicate that only a small number of key - value pairs would have to be maintained before the larger style lock should be applied. Furthermore, the simulation results indicated that subsetting the transactions by less dense attributes (ones with only a handful of different values) would not be beneficial in a lock hierarchy. Thus, in these cases, keeping the predicate locking mechanism quite simple would be justified.

However, such a simple predicate locking mechanism is not very different from a simple physical lock hierarchy. For example, locking a logical relation may in some implementations be identical to locking a physical file or area. At the finest granularity, a predicate lock of a unique key-value pair identifies one record. A physical lock, on the other hand, would uniquely identify the same record by a physical address. Thus, in terms of parallelism and operation, a simplified predicate locking scheme is identical to a physical locking scheme. The physical locking scheme, however, may be easier to implement. Moreover, the physical address for a record might take up less space in a lock table than a predicate lock for the same record.

Another problem exists with predicate locking. In some applications, a secondary key or index is used to access a given record type. Under the simple predicate locking hierarchy described above either all access via a primary key would have to wait for the secondary index application to complete; or the record would have to be read, the key value obtained, and then a lock requested. When the lock is granted the record would have to be reread. With physical locks, on the other hand, the physical record address would be unique.

In summary, then, while predicate locking may be viable, it does not seem to be worth the extra implementation and locking overhead, because it can only be applied when specific sets of the database need to be locking. Furthermore, those cases can be handled adequately by similar physical locking schemes.

CHAPTER 3

DISTRIBUTED DATABASE SYSTEMS

1. INTRODUCTION

In the previous chapter, simulation models were used to investigate the performance issues of concurrency control in a centralized database. In this chapter, those simulation models are extended to study the performance issues of concurrency control in a distributed database.

1.1. Distributed Databases

In a distributed database, the data is stored at multiple computer sites connected by some type of computer network. In this environment, a transaction originates at one of the computer sites and potentially accesses data at other (or remote) sites as well as at the originating site.

The benefits of a distributed database include the ability to share and access geographically distant data, to exercise some local control over subsets of the database, to provide modular growth and resiliency to the database, and to increase the potential parallelism allowed in accessing the database.

1.2. Distributed Database Concurrency Control

The distributed concurrency control mechanism must guarantee the same type of consistency which was needed in the centralized database. However, the performance issues in a distributed database are different than in a centralized database. This difference is due to the following factors:

1) More parallelism is possible because multiple sites can simultaneously process transactions. In the centralized model, at most two servers, the I/O and CPU processors, could be kept busy. In an N site system, there are 2*N servers which can be simultaneously processing transactions.

2) The overhead associated with distributed concurrency control will be higher than the overhead required in a centralized database. The additional overhead is due to the costs required to set locks at remote sites and/or the costs which may be required to resolve deadlock between transactions at different sites. The remote locking overhead is due to the network delays involved with sending and receiving lock messages. The deadlock resolution overhead

includes the computer resources required to detect deadlock and to roll back certain transactions.

The simulation model for the centralized database concurrency control was extended to investigate the trade-offs between the increased parallelism and increased overheads of a distributed database. The major areas of study include the effects of varying the locking granularity, varying the percentage of transactions requiring non-local or remote resources and varying the throughput and bandwidth of the network.

In the next section, the extensions of the simulation model which apply to all distributed concurrency control algorithms are discussed. In section 3, four different concurrency control algorithms and their associated simulation extensions are discussed. In section 4, the simulation results for each of the four algorithms are reported. In the final section, the major conclusions are stated.

2. MODEL EXTENSIONS

In this section the model extensions are described. First, the network model is reviewed. Next the actions at each of the nodes or network sites is discussed. Finally, the input and output parameters of the model are discussed. Throughout this section, only the processing of transactions will be considered. In the next section, four concurrency control algorithms will be integrated into the model.

2.1. Network Model

The network is considered to be a collection of computer sites called nodes, all connected by a "logical network manager" as shown in figure 3-1. This manager could represent a specific star like network, or a more general node to node network like the ARPANET [KLEI76]. In either case it is assumed that the time to send a message between any pair of nodes is the same.

Each Node contains a message-in and a message-out queue. Messages are taken from the message-out queue and given to the Network Manager together with a destination and a message length. When a message has received the needed amount of network service, it is placed on the destination message-in queue.

Both a speed and a bandwidth are associated with the Network Manager. The network speed is represented by the minimum time a message of any type must spend in the network where time is measured in the time units of the simulation. The bandwidth is represented by the maximum

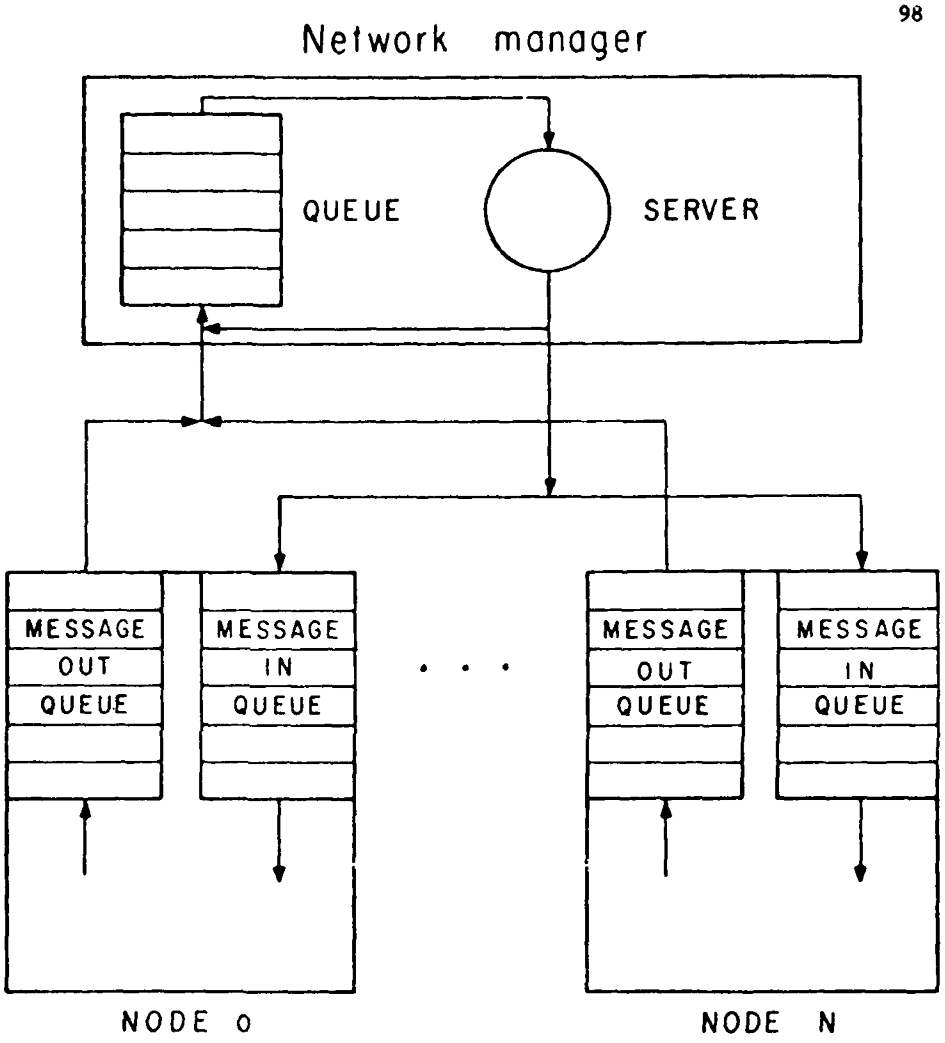


Figure 3-1: Network Model

number of messages which can be serviced in one of those time units.

The flow of a message in the Network Manager can be described as follows:

1) When a message enters the network manager, the time remaining for that message is initialized to the message length in the time units of the simulation. The message length can vary depending on whether or not data is being sent but is at least equal to the minimum length mentioned above. More details on this length are in section 2.3.

2) If MESSBDWH is the bandwidth of the Network Manager, the times remaining of the first MESSBDWH messages in the Network queue are reduced by one time unit.

3) If the time remaining for any message is zero, it is delivered to the message-in queue of the destination node.

In several of the concurrency control schemes, a site can send messages to itself. In these cases, no network resources are consumed or network delay realized, since the message is taken directly off of the message-out queue and placed on the message-in queue. However, local message costs (CPU time spent by a node handling messages) are included for these self-directed messages.

2.2. Site Model

Each site or node in the model is very similar to the centralized model presented in Chapter 2. However, several new queues and procedures were added to process distributed transactions. The new model is shown in figure 3-2. Again transactions are cycled around a closed loop model and initially arrive one time unit apart on the pending queue.

There are three possible types of transactions in the model. First, there are local transactions which are identical to the transactions in Chapter 2. Secondly, there are MASTER transactions which require access to parts of the database at randomly selected other nodes. The MASTER transactions initiate a fixed number of SLAVE transactions at those other nodes via messages.

The transactions go through the following steps: 1) leave the pending queue, 2) I/O processing, 3) CPU processing, 4) data transmission, 5) local processing completion, and 6) distributed processing synchronization.

1) When a transaction leaves the pending queue it is placed on the I/O queue. If the transaction is a

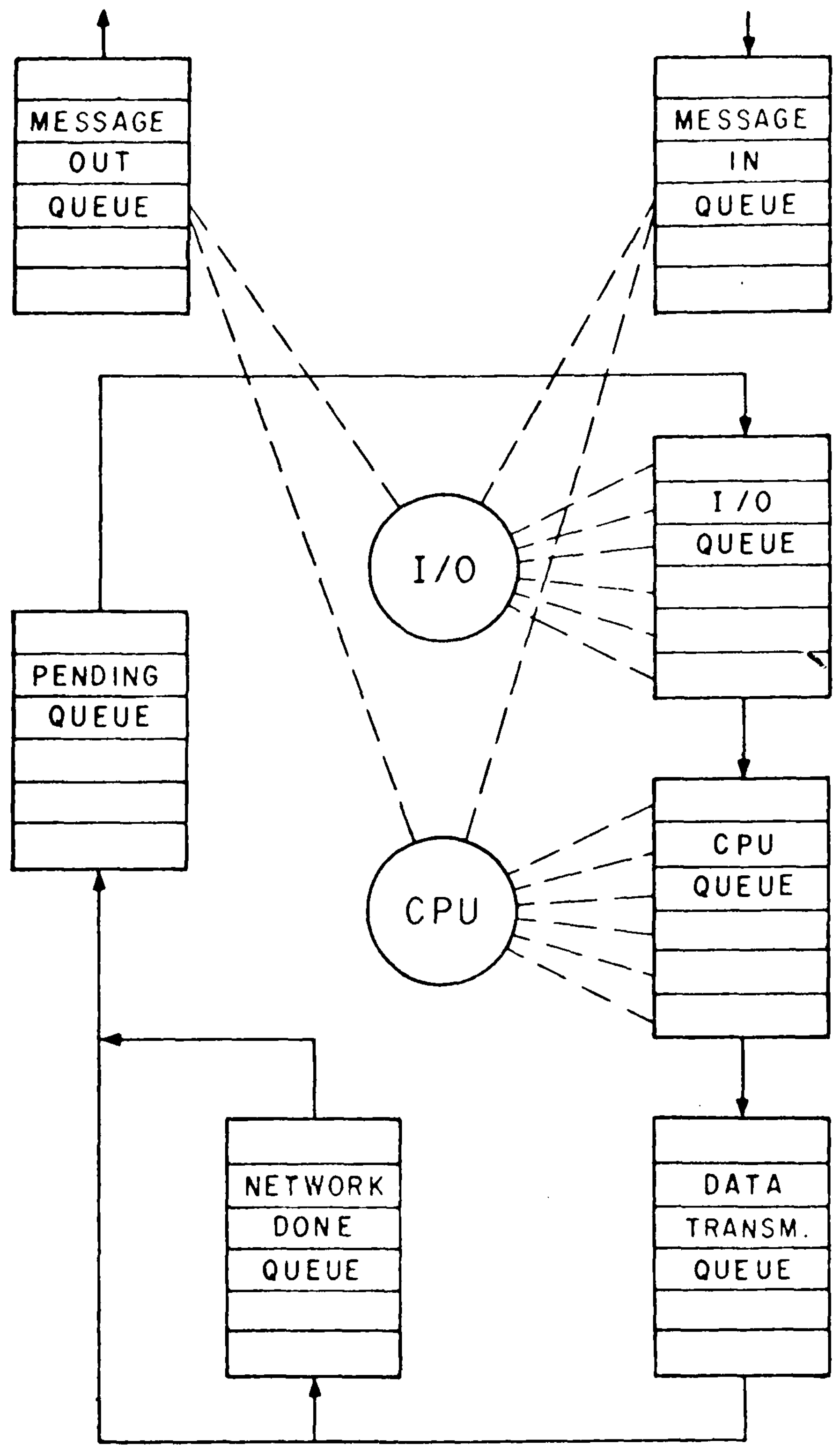


Figure 3-2: Node or Site Model

MASTER, it sends SLAVE create messages to the appropriate nodes.

2) The I/O server is multiplexed among the transactions on the I/O queue. When a transaction has received its share of I/O resources, it is placed on the CPU queue.

3) The CPU server is multiplexed among the transactions in the CPU queue. When a transaction has received its share of CPU resources, its next action depends on whether or not the transaction is local.

4) Local transactions are considered complete at this point and recycled to the pending queue. Non-local transactions (both SLAVES and MASTERS) are placed on the data transmission queues. If any data is to be transmitted, a data transmission message is sent. This transmission message is in fact addressed back to the sending transaction. Thus the data transmission is complete when this message is delivered back to the originating site.

5) When the data transmission message has been received (or if no data was to be transmitted), the non-local transaction proceeds to the Network done queue. At this time, SLAVE transactions send a SLAVE complete

message back to the MASTER transaction.

6) Depending on the concurrency control strategy, a SLAVE either waits on the Network done queue or is simply released. The release of a slave is discussed in more detail in section 3. The MASTER transaction waits on the Network done queue until it has received "slave complete" messages from all its slaves. At that point, the transaction is recycled back to the pending queue.

Three types of messages are common to all of the concurrency control algorithms. The actions caused by these messages are described below.

1) When a "SLAVE create" message is received, a transaction identical to the MASTER transaction, only flagged as a SLAVE is added to the pending queue.

2) When a "data transmission done" message is received, the waiting MASTER or SLAVE transaction is notified.

3) When a "SLAVE complete" message is received, the corresponding MASTER transactions on the Network done queue is notified. If the MASTER transaction is not completed, the message is returned to the message-in queue until the MASTER transaction completes.

Several simplifying assumptions should be noted about the model. First, all of the SLAVEs are identical to the originating MASTER in terms of the proportion of database accessed and whether or not data is to be transferred. In distributed database applications, the actual characteristics of the SLAVEs could be quite different from the MASTER and from each other. Second, the only synchronization between the SLAVEs and their MASTER transaction occurs at the beginning and end of the transaction. Some applications would require additional synchronizations on the data being transmitted [WONG77, EPST78].

Also note that a transaction is on each of the I/O, CPU and data transmission queues once in the indicated serial order. The total processing required is the same as if the transaction cyclically accessed the I/O, CPU and data transmission queues.

2.3. Model Parameters

The input parameters can be divided into the parameters that characterize the workload, the system parameters that characterize the individual nodes, and the parameters that characterize the network. The workload parameters determine the database and the transactions that are run against that database. As in Chapter 2, the system parameters determine the computer and database management system characteristics. The network parameters include the minimum time required for messages, the network bandwidth and the CPU and I/O resources required for processing messages at each site.

The output measurements include the overall CPU and I/O resource utilizations for transactions, messages and concurrency control as well as network measurements.

These parameters, in most cases, have the same interpretation as in Chapter 2. All of the parameters are described in detail below.

2.3.1. Workload Parameters

The workload parameters describe the transactions and the portion of the database at each node. Table 3-1 summarizes the workload parameters.

The first five parameters are identical to the parameters discussed in Chapter 2. For almost all of the experiments reported in this chapter only a few settings of those parameters are used. The effects of varying those parameters would be similar to the effects reported in Chapter 2.

In particular, NTRAN was set to 10 simulating 10 transactions running at each node. The DBSIZE at each

Table 3-1
Workload Parameters

Parameter	Description
	Local Parameters
NTRAN	Number of transactions running at each node
DBSIZE	Size of the portion of a database at a given node
AMEAN	Low-mean of exponential distribution for transaction size
BMEAN	High-mean of exponential distribution for transaction size
ALPH	Cut point for Hyper-exponential distribution for transaction size
LKPLMT	Lock Placement assumption
	Distributed Parameters
PREDIST	Percentage of transactions which are non-local
PRETRAN	Percentage of distributed transactions which transfer data
PREDATT	Percentage of data transferred by those distributed transactions
NSLAVES	Number of SLAVES for a distributed transaction

node was set to 10,000, resulting in a total database size of 10,000 times the number of nodes in the network.

Two classes of transactions are modeled. With class one transactions, the transaction sizes vary considerably and the locks are assumed to be well-placed with respect to the accessing transactions. For these transactions AMEAN is 5, BMEAN is 250 and the ALPH parameter was set to .1. This class of transactions simulates a workload where most (90%) of the transactions are small (they access 0.05 percent of the database) and a few of the transactions are large.

With class two transactions, all transactions are small and the placement of the locks is assumed to be random with respect to the accessing transactions.

The remaining parameters deal with the non-local transactions and are the ones of most interest in this chapter. The proportion of transactions which are MASTERs, the PREDIST parameter, determines the number of transactions at each node which require processing at some other site. Experiments were run with PREDIST settings of 0, 10, 25, 50, 75 and 100 percent.

The number of SLAVES required by a MASTER are determined by the NSLAVES parameter. The original number of database entities required by the MASTER is evenly distributed among the SLAVEs and the MASTER. Thus, if the MASTER originally requires E of the database entities, at each site where the transaction was active, E/(NSLAVES + 1), entities are actually accessed.

The amount of data to be transferred is determined by the PRETRAN and PREDATT parameters. The PRETRAN parameter determines the number of distributed transactions which transfer any data at all. The PREDATT parameter determines how many of a transaction's entities will have to be transferred. The number of entities transferred determines the length of a data transfer message and hence determines how long a transaction spends on the Network wait queue.

In summary, the database consists of a collection of entities at each node. Each transaction "touches" or accesses a certain number of those entities. Some of those transactions require access to entities at remote nodes. Furthermore, some of those transactions will have to transfer data between nodes.

2.3.2. System Parameters

The system parameters describe the computer system or database system at each node and are very similar to the system parameters of the centralized database model described in Chapter 2. The system parameters are summarized in Table 3-2.

The NGRAN parameter is the number of lockable granules at each node of the distributed database and is identical to the NGRAN parameter of Chapter 2. The param-

Table 3-2 System Parameters

Parameter	Description
NGRAN	number of lockable units of one node of the database
CPURATE	CPU time to process one entity
IORATE	I/O time to process one entity
LCPURATE	CPU time to process one lock
LIORATE	I/O time to process one lock

eter was varied from 1, representing one lock at each node, up to DBSIZE, representing one lock per entity in the database.

The CPURATE and IORATE determine the cost to process one entity in the database and are also identical to the parameter in the centralized database model. For these experiments, the CPURATE and IORATE were both equal to 1 time unit. This scenario simulates a system with a balanced load between the CPU and I/O requirements. Also under this scenario, one time unit of the simulation can be thought of as the time required for one I/O operation, i.e., about 30 milliseconds.

The LCPURATE and LIORATE parameters, the costs to set and release one lock, are also identical to the parameters in the centralized database model. For these experiments, the lock CPU rate was one tenth the entity CPU rate, i.e., 0.1. Under the scenario mentioned above, this might represent 3 milliseconds to set and release a lock. The LIORATE was zero, simulating a system where all locks are kept in main memory.

Note that NGRAN, LCPURATE, LIORATE and [illegible] (from the previous section) are locking parameters used by all of the concurrency control algorithms. Additional parameters relevant to the individual concurrency control algorithms are introduced in the section describing those algorithms.

2.3.3. Network Parameters

The network parameters determine the throughput and bandwidth of the network as well as the CPU resources required at each site to send and receive a message. The network parameters are summarized in Table 3-3.

The number of nodes in the network, set by the NNODES parameter was varied from two to eight.

The message rate parameter, MESRATE, is the length of time it takes to send a simple message (i.e. a non-data transfer message) from one node to another. Typical values for this parameter ranged from 1 through 10. A value of 3, for example, would represent a high speed network, where, under the interpretation mentioned in the previous section, it would take about 90 milliseconds to

Table 3-3 Network Parameters

Parameter	Description
NNODES	The number of nodes or sites in the network
MESRATE	The time units a message must stay on the network
DATARATE	The time units to transfer an entity
MESBDWT	The number of simultaneous messages or bandwidth of the Network Manager
MESIORATE	The I/O time required by a node to send or receive a message
MESCPURATE	The CPU time required by a node to send or receive a message.

deliver a message. A value of 10 implies it would take about 300 milliseconds to send a message, about the time required on the ARPANET [KLEI76].

The DATARATE, together with the MESRATE parameter, determines how long a data transmission message will take. If E is the number of entities to be transmitted, then the data transmission message would take

MESRATE + E * DATARATE

time units to be delivered. If an entity is a 512 byte page, and an ARPANET like file transfer at 50,000 bits per second is assumed, it would take about 0.1 seconds to transfer 1 entity. On the other hand, or a three megahertz speed network, it would only take about .0015 seconds. Many of the simulation experiments used and "optimistic" DATARATE of .05 time units. Other simulation experiments used DATARATES of .1, .25, and .5. The MESRATE term is included in the above time to represent the initialization message which often must precede a network data transmission.

The MESBDWT parameter determines the bandwidth of the network manager. As explained in section 2.2, at most MESBDWT messages in the network queue are serviced each time unit. For lightly loaded networks, it is reasonable to assume that the bandwidth is unbounded [KLEI76] and that assumption is in fact made in most of the simulation experiments. The results of varying that parameter are also presented.

The MESIORATE and MESCPURATE parameters represent the resources required at each node to send or receive a message. For these simulation results, the MESIORATE was zero, simulating that the processing of all messages is handled in the main memory; and the MESCPURATE has a value ranging from .01 to .3 time units; or in the canonical interpretation from .3 to 9 milliseconds. For the most part, the lower bound on MESCPURATE was used, simulating a very low (and optimistic) overhead message processor.

2.3.4. Output Parameters

The quantities to be measured or the output parameters are summarized in Table 3-4. These measurements include all of the measurements included in the centralized database simulation and some other parameters unique to the distributed model.

The first eight output parameters are identical to the output parameters discussed in Chapter 2. The TCPU and TIO parameters refer to the number of simulation time units during which the CPU and I/O servers for all of the network nodes were kept busy. The LOCKCPU and LOCKIO

Table 3-4 Output Parameters

Parameter	Description
	Local
TCPU	Total time the CPU server was active
TIO	Total time the I/O server was active
LOCKCPU	CPU overhead for locking
LOCKIO	I/O overhead for locking
TRANCOM	Number of transactions completed
AVERRES	Average response time
USEFULCPU	CPU time for processing transactions
USEFULIO	I/O time for processing transactions
	Distributed
MESCPU	CPU overhead for network messages
MESIO	I/O overhead for network messages
TMESS	The total number of messages sent
LMESS	The number of Lock related messages sent

parameters refer to the number of time units the respective servers were busy managing locks. The TRANCOM parameter is the total number of transactions completed at the end of a simulation run. Note that a distributed transaction, regardless of the number of corresponding SLAVE transactions, is counted as one transaction. The AVERRES parameter measures the average number of time units it takes for a transaction to complete. For distributed transactions, the response time refers to the time difference between when a MASTER transaction first enters the pending queue and when that transaction leaves the network done queue.

The useful computer utilizations, USEFULCPU and USEFULIO, refer to the resources used for transaction processing. These measurements were not used for concurrency control or for message processing. Note that

TCPU=USEFULCPU+LOCKCPU+MESCPU

TIO=USEFULIO+LOCKIO+MESIO.

The MESIO and MESCPU parameters refer to the time required by the I/O servers and CPU servers at the various nodes to process messages. Note that a message must both be sent and received, so that the I/O and CPU costs to send n messages are n*2*MESIORATE and n*2*MESCPURATE respectively. This cost is also independent of the message length. Thus, for a data transfer message, this cost represents initial set up costs to actually transfer data to the network. No additional local costs for the data transfer are incurred. In some systems considerably more overhead would be incurred for data transfer.

The TMESS parameter represents the total number of messages sent over the network. The LMESS parameter is the number of those messages which were sent only for concurrency control reasons. The messages, called 'Lock' messages, are discussed when the concurrency control algorithms are introduced.

2.4. Typical Scenarios

The simulations were run with the parameter settings shown in Table 3-5. The local parameters are identical to the parameters in the centralized database simulations and for the most part were not varied. The initial settings of the distributed parameters are designed to study the concurrency control algorithms under a basically free and unlimited network. Later alternate parameter settings are used to study the effects of network limitations on the different concurrency control algorithms. The results of those experiments are reported in section 4.

In the next section, the four concurrency control algorithms simulated are described and additional parameters required for those algorithms are introduced.

3. DISTRIBUTED CONCURRENCY CONTROL

The distributed database concurrency control algorithms can be divided into two general classes: primary site concurrency control [ALSB76, MENA78] and decentralized concurrency control [STON78, ELLI77, GRAY78, [illegible]].

In the primary site concurrency control schemes for a distributed database, one site is chosen to enforce a processing schedule equivalent to a global serialization of

Table 3-5 Typical Parameter Settings

Parameter	Setting	Interpretation
	Local	
NTRAN	10	10 trans at each node
DBSIZE	10,000	10,000 database entities at each node
AMEAN	5	0.05% of DBSIZE
BMEAN	250	2.5% of DBSIZE
ALPH	.1	10% of trans are large
	0	All trans are small
LKPLMT	1	Well-placed locks (used with ALPH = .1)
	2	Randomly placed locks (used with ALPH = 0)
CPURATE	1	30 msecs
IORATE	1	30 msecs
LCPURATE	.1	3 msecs
LIORATE	0	Locks in main memory
	Distributed	
PREDIST	.1	10% of the transactions are distributed
PRETRAN	.40	40% of those require data transfer
PREDATT	.25	25% of entities touched by the transactions are in fact transferred
NSLAVES	5	A distributed transaction runs at all nodes
NNODES	6	Six nodes in the network
MESRATE	3	High speed network
DATARATE	.05	fast data transfer Network
MESRCNT	∞	Lightly loaded network
MESIORATE	0	Messages handled in core
MESCPURATE	.01	.3 msecs (very optimistic)

all of the transactions running at all sites. Two straightforward implementations of a primary site model are presented in sections 3.1 and 3.2. Basically, if a primary site handles all concurrency control, the same algorithm used for a centralized database can be used for the distributed database.

In the decentralized concurrency control schemes, each site maintains its own locks for that site's portion of the database. However, a deadlock condition [COFF71] can exist in the network even though no deadlock cycle exists at any given node. For example Transaction 1 can be blocked at node i by Transaction 2. At node j, however, Transaction 2 can be blocked by Transaction 1. Although no deadlock exists at either node i or j, neither Transaction 1 nor Transaction 2 can be completed. Two mechanisms and their simulation implementations for dealing with this deadlock problem are presented in sections 3.3 and 3.4.

3.1. Primary Site Model 1

The concurrency control mechanisms in both the primary site models require the following changes to the node model shown in figure 3-2:

1) When any transaction (local or MASTER) leaves the pending queue, a global lock request is sent to the node selected as the "PRIMARY" site. The transaction then waits on a new queue, the global pending queue, until all of its locks are granted.

2) When a global lock grant is received, the transaction can proceed to the I/O, CPU and data transmission queues as before. At this time, a MASTER transaction starts its corresponding SLAVE transactions.

3) Upon receipt of a "SLAVE create" message, a new transaction identical to the MASTER transaction is placed directly on the I/O queue.

4) As in section 2.2, a MASTER transaction waits on the Network done queue until it has received "SLAVE complete" messages from each of its SLAVEs. At this point, a MASTER transaction sends a "global lock release" message to the PRIMARY site and is recycled back to the pending queue.

5) In the PRIMARY site model, a SLAVE transaction need not wait on the Network done queue; it can simply send its "SLAVE complete" message and leave the system.

Note that the "global lock request", "global lock grant", and "global lock release" messages are all included in the lock message count. Also note that the "global lock request" includes the lock requests for all of the SLAVEs.

In the primary site model, the nodes are considered to be numbered zero through NNODES - 1. For each node, there is a "blocked" queue and a "locks held" queue as shown in figure 3-3.

When a "global lock request" is received, the PRIMARY site lock controller goes through the following steps:

1) Determine which nodes will be used by the requesting transaction.

2) For each node, i = 0,...,NNODES - 1, see if this transaction requires locks; if not, proceed to the next node. If so, request the locks (identical to a lock request in Chapter 2) required at this node.

3) If the locks are granted, record this fact on the "locks held" list for node i and repeat step 2 for nodei+1.

4) If the locks are denied, place the transaction on the blocked queue for node i, recording the blocking transaction which is on the "locks held" queue for this node.

5) When the locks at all of the required nodes are granted, a "global locks granted" message is sent to the originating site.

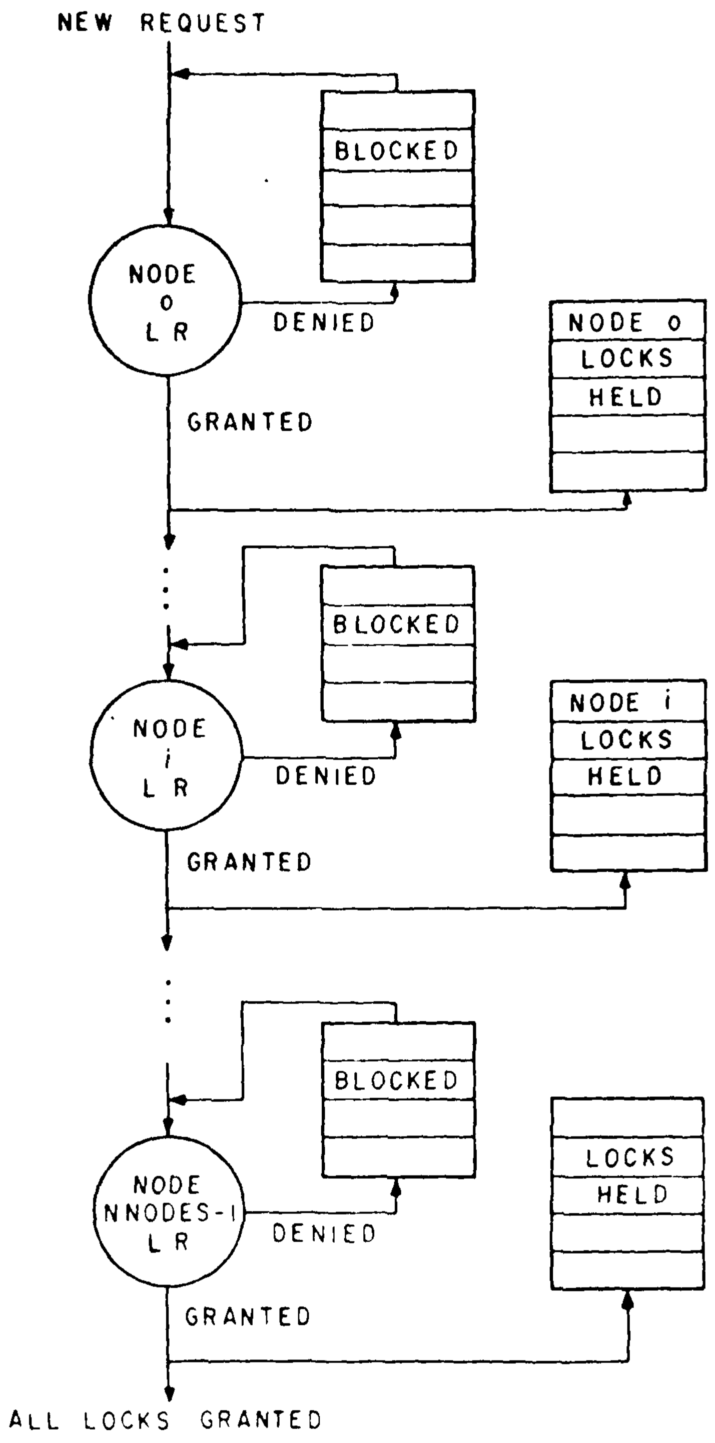


Figure 3-3: Primary Site Model

When a "global lock release" is received, the PRIMARY site lock controller removes the corresponding transaction from each of the "lock held" queues. Any transaction which was blocked at node i by this transaction is restarted at step 2 for node i, in the above algorithm.

The following observations should be noted. First, deadlock is impossible, since the locks at the different sites are always acquired in a fixed order. Second, LOCAL transactions will only be involved with locks at their originating sites. Third, note that a non-local transaction can wait for locks at one node while holding locks at a lower numbered node.

All of the locking costs are absorbed by the primary site which also has a normal load of transaction processing. The use of the CPU and I/O servers by the primary site control mechanism has a preemptive priority over transaction processing requests. In other words, if there are global lock releases or requests, the PRIMARY site first has to serve those requests before it can process any transactions. If serving those requests takes more than one simulation time unit, no transaction processing takes place during that time unit.

3.2. Primary Site Model 2

The activities at each site of the distributed database are identical under this model and the previous primary site model described above. The only difference in the two models occurs in the Primary site lock control. In this model, there is only one blocked queue, although there is a 'locks held' queue for each node.

When a "global lock request" is received, the PRIMARY site lock controller goes through the following steps:

1)-3) Same as in previous primary site model.

4) If the locks are denied, release all of the locks held for lower number nodes, record the blocking transaction and place this transaction on the single blocked queue.

5) Same as in the previous primary site model.

When a "global lock release" is received, the PRIMARY site controller again releases the locks held at each node. Any transaction which was blocked, is restarted at step 1 of the above algorithm.

This model differs from the previous model in two ways. The main difference is that no transaction can hold locks at one node while waiting for locks from another node. This difference means that transactions requiring a fewer number of nodes, (i.e., local transactions) have an implicit priority over transactions requiring locks at more nodes.

3.3. Wound-Wait Model

As previously mentioned, decentralized concurrency control requires a mechanism for resolving deadlock. In this section an extension of a "wound-wait" scheme [ROSE77] for resolving deadlock is discussed. First the original algorithm in [ROSE77] is presented, followed by two extensions. Finally, additional changes in parameters, relevant to the "wound-wait" algorithm are reviewed.

3.3.1. Original Wound-Wait Algorithm

In [ROSE77], the transaction model is slightly different than the one presented in this chapter. A transaction is viewed as a process which is initiated at one node and moves from node to node in the course of its processing. At any instance the process is considered active at one node and inactive at all other nodes that it has visited.

Under the wound-wait algorithm, a unique number, assigned to each process or transaction, is obtained by concatenating a starting time with the node number at which the process is initiated. (The algorithm does not require that the clocks which generate the time stamps be perfectly synchronized. However, some close correspondence with the "real" time would be desirable. In [LAMP77], a sufficient algorithm for keeping clocks at nodes in a network reasonable synchronized is presented.)

Suppose Transaction 1 requests locks held by Transaction 2 and that timestamp 1 and timestamp 2 are the unique numbers associated with the two transactions. Then the following steps are taken:

1) If timestamp 1 < timestamp 2, then Transaction 1 is "older" than Transaction 2. In this case, Transaction 2 is wounded and Transaction 1 waits.

2) If timestamp 2 < timestamp 1, then Transaction 2 is "older" than Transaction 1. In this case, Transaction 1 simply waits.

If Transaction 2 is wounded, a message is sent to all sites visited by Transaction 2. If termination of Transaction 2 has already begun, the wound is ignored, since Transaction 2 will soon release its locks and Transaction

1 can proceed. If Transaction 2 has not begun the termination process, it is aborted (or killed) and restarted. Again, the locks held by Transaction 2 are released and Transaction 1 can proceed. Note that in order to prevent cascading abortions of transactions, all locks for a given transaction are held until that transaction terminates.

A natural modification to this algorithm is suggested in [ROSE77], where Transaction 2 is not aborted and restarted unless it is actually in or enters a waiting state.

This algorithm provides a consistent concurrency control for which every transaction terminates. Consistency is maintained because a transaction holds all locks until it has completed. Thus, two-phased locking is insured. To see that every transaction terminates, note that at any given time, due to the uniqueness of the timestamp, there is exactly one "oldest" transaction. That transaction can never be wounded and thus must terminate. At that point, there is a new "oldest" transaction which also must terminate. A transaction retains its original timestamp even if it is restarted.

3.3.2. Simulation Implementation

To apply the above algorithm to the distributed transaction processing discussed in this chapter, it must first be noted that a wounded transaction can be active at more than one site. Thus, the decision to abort and restart a transaction might be initiated at several sites. A wound or kill message for an already killed transaction is simply ignored. When a transaction is restarted a 'cycle number' is incremented. The cycle number, initially zero, is included in the message addresses so that the restarted transaction does not erroneously receive an old wound or kill message.

A second simple modification to the [ROSE77] algorithm was also made. A transaction receiving a wound message is not restarted unless that transaction is blocked by or becomes blocked by a transaction that the original transaction cannot wound. In other words, a wounded transaction must be restarted if and only if it is blocked by an older transaction. Note that this algorithm still resolves any potential deadlock and all transactions must eventually terminate.

Theorem: This modified wound-wait system still preserves consistency and every process terminates.

Proof: The database consistency is preserved since the locking is still two-phased.

Every process will terminate, since a deadlock cycle cannot exist in the wait-for graph. The wait-for graph is a directed graph where the nodes represent transactions in the system. An arc from one node to another implies that the first node represents a transaction that is blocked by the transaction represented by the second node.

Suppose a deadlock cycle existed in the graph between nodes $T_1, \ldots, T_n$ (i.e., T_1 is blocked by T_2, T_2 is blocked by T_3, ..., T_{n-1} is blocked by T_n, and T_n is blocked by T_1). Without loss of generality, assume T_1 is the oldest transaction. Then T_1 must wound T_2.

If T_2 can wound T_3, it does. If not, T_2 is aborted and the deadlock no longer exists.

Similarly, if any T_i cannot wound T_{i+1}, it must be aborted.

If all of the T_i ($i = 2, \ldots, n$) are wounded, so is T_n.

But T_n is blocked by T_1 and cannot wound T_1 because of our assumptions.

Thus, T_n must be aborted and restarted and the deadlock cycle is broken.

Q.E.D.

To implement the above algorithm, several modifications to the simulation were made. First, the arrival time of each transaction was guaranteed to be unique by insuring that all transactions arrived at least one time unit apart. The arrival time, concatenated with a node number, in the least significant bits, is the unique timestamp associated with the transaction. In addition, a cycle number is added to each transaction in order to insure that a restarted transaction is not wounded or killed by a message intended for an earlier incarnation. When a transaction (SLAVEs excepted) is first placed on the pending queue, the cycle number is initialized to zero. Messages are only delivered to transactions with the correct cycle numbers. Messages destined for earlier cycles are simply discarded. A SLAVE transaction takes on the timestamp and cycle number of its corresponding MASTER. The following steps are now followed by a transaction.

1) A transaction leaves the pending queue. If the transaction is a MASTER and this is the first time this incarnation has left the pending queue, "SLAVE create" messages are sent to the appropriate nodes.

2) After leaving the pending queue, a transaction requests the locks needed at that site. If the locks are granted the transaction proceeds to the I/O, CPU and data transmission queues. If the locks are denied, the transaction is placed on the blocked queue for this node. Let T1 be the requesting tran-

saction and T2 be the blocking transaction respectively. If both T1 and T2 are distributed transactions (SLAVES or MASTERS) and T1 is older than T2, transaction T2 is "wounded". If T1 is younger than T2 and has been previously wounded, T1 is killed.

3) Once the locks are granted, the transactions proceed on the I/O, CPU and data transmission queues as before.

4) A SLAVE transaction sends a "SLAVE complete" message to its MASTER and waits on the Network done queue for a release locks message. A MASTER transaction waits on the Network done queue until all of its SLAVEs have completed.

5) When all of the SLAVEs have completed, the MASTER sends a "release locks" message to all of its SLAVEs, releases its locks, and becomes a new transaction on the pending queue. At this point the transaction is considered done. When a SLAVE receives the "release locks" message, it releases its locks and leaves the system.

6) When any transaction releases its locks, the corresponding blocked transactions (if any) are placed at the front of the pending queue.

When a transaction is wounded, "wound" messages are sent to the MASTER and all SLAVEs. When a transaction receives a wound, it is flagged as wounded. If the transaction is already blocked by a distributed transaction with an older timestamp, the wounded transaction is immediately "killed".

When a transaction is killed, "kill" messages are sent to the MASTER and all SLAVEs. Both types of transactions release their locks and blocked transactions are placed on the front of the pending queue as in step 6 above. Any time spent on the I/O or CPU queues is counted in a "lost time" total. At this point SLAVE transactions leave the system. A MASTER transaction increments its cycle number and is placed on the back of the pending queue for a reincarnation.

A few observations should be made. First, in step 2, only if both transactions are non-local, does a potential wound have to take place. If the blocking transaction is local, it is guaranteed to finish since it has preclaimed all of its locks. If the blocked transaction is local, it can hold no locks at other sites and thus no deadlock can occur.

Second, when a transaction is restarted, it is placed on the pending queue behind any transactions that it blocked. In particular, it is placed behind the transaction that caused the original wound. Thus, the same wound will not occur again.

Note that the extra "release lock" messages to the SLAVEs are not present in the primary concurrency control models. They are not needed in those models because all locks are held (and thus released) at the primary site.

3.3.3. Additional Parameters

Four additional output parameters were recorded in the simulation model for the "wound-wait" concurrency control. In addition, new types of messages are classified as lock messages.

The four output parameters are for the number of transactions wounded (NTRWOUND), the number of transactions killed (NTRKILL), and the lost time attributed to killed transactions (DLOSTIO and DLOSTCPU). The count of the number of transactions wounded, NTRWOUND, is made only when a cycle of a given MASTER receives its first wound. Thus, even though the SLAVEs all receive wound messages, the wounding of a distributed transaction is only counted once.

Similarly, the count of the number of transactions killed, NTRKILL, is made only when a MASTER receives its first kill.

The DLOSTIO and DLOSTCPU parameters record the number of time units of I/O and CPU service respectively that a killed transaction has received. How much time is actually lost depends on both the queue the transaction is on and the processing completed at that queue. Note that the definition of the total I/O utilization, TIO and the total CPU utilization, TCPU also changes:

TIO = USEFULIO + MESIO + LOCKIO + DLOSTIO

TCPU = USEFULCPU + MESCPU + LOCKCPU + DLOSTCPU.

In the "wound-wait" concurrency control algorithm for a decentralized database, "WOUND", "KILL", and "lock release" messages are all counted as lock related messages. The lock related messages used in the primary site models are no longer relevant.

3.4. SNOOP Model

A second decentralized concurrency control algorithm uses a 'SNOOP' [STON78] or a global deadlock detector [GRAY78] was also simulated. One problem with the "wound-wait" algorithm is that transactions may be killed and restarted needlessly. While the algorithm is sufficient to prevent deadlock, it may be too conservative. Transaction 1 could be blocked by the younger Transaction

2. Transaction 2 could be blocked by the older Transaction 3, which can in fact complete. Even though no deadlock is present, Transaction 2 would still be restarted.

In this section, an algorithm is described which restarts transactions only when an actual deadlock occurs. In section 3.4.2, the implementation of that algorithm in the simulation model is presented. In section 3.4.3 changes in the simulation parameters are discussed.

3.4.1. SNOOP Algorithm

In [STON78] a decentralized algorithm for concurrency control is presented. Each node or site in the distributed database is responsible for local concurrency control for the portion of the database at that site. If two transactions conflict, the local concurrency control sends a message about this conflict to a designated site called 'The SNOOP'.

The SNOOP then detects deadlock by an analysis of the "wait-for" graph generated by all such messages. If a deadlock condition is detected, a victim is picked to be killed and restarted (a reincarnation). Note that when a transaction has completed, the SNOOP must also be notified so that the appropriate entries in the "wait-for" graph can be cleared.

The same basic idea was also suggested in [GRAY78] with several modifications. One modification is that a conflict message is only sent to the SNOOP if the blocking transaction is directly waiting on a response from another node or is blocked (directly or indirectly) by some other transaction that is waiting on a response from another node.

Another suggested modification is to only send such conflict messages and check for deadlock periodically. In this manner the system overhead for both handling lock messages and checking for deadlock can be reduced at the cost of delaying the detection of an existing deadlock.

3.4.2. Simulation Implementation

The 'SNOOP' simulation model is very similar to the wound-wait model. The following steps are taken in the 'SNOOP' model.

1) Same as "wound-wait" model

2) Same as "wound-wait" if the locks are granted. Suppose the locks are denied and T1 is the requesting transaction and T2 is the blocking transaction. If both T1 and T2 are distributed transactions (SLAVES

or MASTERS), a conflict message is sent to one of the sites designated as the SNOOP.

3,4) Same as "wound-wait" model.

5) Same as "wound-wait" except that when a MASTER transaction is done, a "clear snoop" message is sent to the SNOOP.

6) Same as "wound-wait" model.

The SNOOP maintains a global "wait-for" directed graph. Each node represents a blocked or blocking transaction. An arc from node 1 to node 2 implies that the transaction represented by node 1 is blocked by the transaction represented by node 2. When a conflict message is received, a node for each transaction (if one doesn't already exist) is added to the graph along with the appropriate arc. At that point, the graph is searched for a cycle beginning at the node for the blocked transaction. If deadlock is detected, the youngest (determined by the unique timestamp) of the two transactions involved with this conflict is declared a victim and killed. The fact that a given cycle of the victim was killed is remembered by the SNOOP.

The killing of a transaction is identical to the killing of a transaction in the wound-wait algorithm. The SNOOP sends a message to the MASTER and its SLAVES. Both types of transactions release locks and record lost time. A MASTER transaction is reincarnated as in the wound-wait model.

Note that it is necessary that the SNOOP remembers both killed and completed transactions for a given period of time. It is possible that the SNOOP could be notified of a conflict involving a killed, or completed transaction. In these cases, the conflict occurred before a node received the 'kill' or 'release locks' message. In the case of a killed transaction, a false deadlock could be detected. In the case of a completed transaction, an extra node would simply clutter the wait-for graph. If a killed or completed transaction is involved in a conflict message, the message is simply ignored at the SNOOP site.

The cycle number is needed by the SNOOP to distinguish between messages meant for different incarnations of a transaction. If a conflict message arrives with a higher cycle number than the cycle number of a killed node, the killed node is removed and a new node inserted in the graph. If a conflict message arrives with a lower cycle number than the corresponding node in the graph, the message is simply discarded. Such messages are obsolete.

Also note that in the case of deadlock, the victim is chosen from among the two nodes involved in this conflict. This choice is guaranteed to break any deadlocks since the graph is assumed to be deadlock free before the latest arc was added. This victim may not be the optimum victim for backout. However, if a different victim were chosen, the other parts of the graph would still have to be searched for other deadlock cycles.

Finally note that all conflicts between non-local transactions are sent immediately to the SNOOP. As previously mentioned, it is suggested that the conflict messages should not be sent unless the blocking transaction actually enters a "node wait" state. However, in this model, both MASTER and SLAVE transactions will eventually wait for messages from other nodes before they release their locks. Since that 'node-wait' state is inevitable, the conflict messages are sent immediately.

3.4.3. SNOOP Parameters

As with the "wound-wait" algorithm, several new parameters are introduced and the definition of lock messages is changed.

The cost to check for deadlock is very expensive; often this cost is much greater than the cost to set a simple lock [GRAY78]. A new network input parameter, SNOOPRATE, was added to the simulation to model that additional cost. Every time a conflict message is received by the SNOOP, SNOOPRATE time units are added to the locking costs at the SNOOP node. Note that conflicts involving killed or completed transactions are not included. In most of the experiments, a SNOOPRATE of .5 is used. In the canonical interpretation this value represents about 15 milliseconds or about 5 times the cost to set a lock.

The NTRKILL, DLOSTIO, and DLOSTCPU parameters from the wound-wait model are also included in the SNOOP simulation. However, rather than the NTRWOUND parameter, the SNOOP model records the NUMCONFLCT parameter, the number of actual conflict messages received. Again, conflict messages for already killed or completed transactions are not included in this count.

The lock messages in the SNOOP model are the 'conflict' messages, the 'kill' messages, the 'release locks' messages and the 'clear SNOOP' messages.

4. RESULTS AND DISCUSSION

The results for the distributed database simulations are presented in this section. In the first section we present the results for the parameter settings for the canonical scenarios. Subsequent sections review the effects of varying the number of SLAVES for each distributed transaction (NSLAVES), the number of nodes in the network (NNODES), and the percent of distributed transactions (PREDIST).

In section 4.5, the results of varying the network parameters are repeated. These parameters are the message rate (MESRATE), the network bandwidth (MESBDWT), the CPU rate for processing messages (MESCPURATE) and the percentage of data transferred (PRETRAN and PREDATT). Finally, the canonical cases are revisited in section 4.6 with a different network environment.

The results are reported for each of the four concurrency control algorithms simulated and the two different classes of transaction sizes. The first primary site model, where locks for one site are held while waiting for locks at another site, is denoted "PS1". The second primary site model is denoted "PS2". The notation "WW" refers to the wound-wait algorithm, while "SNOOP" refers to the algorithm with the single global deadlock detector.

Transactions in class 1 refer to transactions whose sizes are generated by a hyperexponential distribution and well-placed locks are assumed. Transactions in class 2 refer to transactions whose sizes are mainly small (generated by an exponential distribution). In this case, random lock placement is assumed.

In the first three sections, an unlimited network is assumed in order to study the effects of the different concurrency control algorithms on the processing at each of the nodes. Beginning in section 4.4, network limitations are introduced to study the effects of the concurrency control algorithm on the network resources.

4.1. The Canonical Scenarios

The canonical scenarios refer to the cases where the input parameters have the settings shown in Table 3-5. For these experiments, as in Chapter 2, the number of locks (NGRAN), was varied from 1 up to DBSIZE and reflects the number of locks at each node. One lock implies that at each node, only one transaction can be active at one time. With 10,000 locks, there is one lock for each entity at each node and transactions can proceed if the entities they require are not being accessed by any other transaction.

Table 3-6 shows the expected number of each type of message under the canonical parameter settings. Each non-local transaction sends 5 (NSLAVE) slave create messages and receives 5 slave completed messages. In addition, 40% (PRETRAN) of the non-local transactions send 6 (NSLAVES +1) data transfer messages. These non-local messages are the same for all four concurrency control algorithms.

However, the four algorithms send different numbers of lock messages. In the primary site models, transactions at 5 of the 6 nodes (all nodes other than the primary site) have three lock messages: "request locks", "grant locks", and "release locks". In the decentralized models, only the non-local transactions send lock messages. Those messages include the 5 (NSLAVE) release lock messages plus some messages for wounding transactions,

Table 3-6
Expected Messages per Transaction

	PS1-PS2		WoW-SNOOP	
	Local	Non-local	Local	Non-local
Non-lock Messages	0	10+(.4)6	0	10+(.4)6
Lock Messages	3(5/6)	3(5/6)	0	5+? (WW) 5+(5/6)+? (SNOOP)

killing transactions and/or notifying the SNOOP of conflicts. In addition, in the SNOOP model, a non-local transaction at other than the SNOOP site must send a "clear SNOOP" message when it has completed.

The results for the canonical scenarios are presented for class 1 and class 2 transactions. Figures 3-4 and 3-5 show the effects of varying the number of locks at each node on the USEFULIO for each of the four concurrency control algorithms. The horizontal axis represents the number of locks in a logarithmic scale. The vertical axis is the USEFULIO, or I/O resources used in completing transactions, in 1000 time units of the simulation. Note that for six nodes, at most 120,000 time units (NNODES*TMAX) of I/O resources are available. The curves for the USEFULCPU measurements were very similar and are not shown.

4.1.1. Class 1 Transactions

Figure 3-4 shows the results for class 1 transactions. For all four concurrency control algorithms, the maximum USEFULIO occurred with 500 to 1000 granules. For the primary site 2 (PS2) and the global deadlock detector (SNOOP) models, the peak occurred at 500 granules. For the primary site 1 (PS1) and wound-wait (WW) models, 1,000 granules were optimal. In either case, with 1% of the

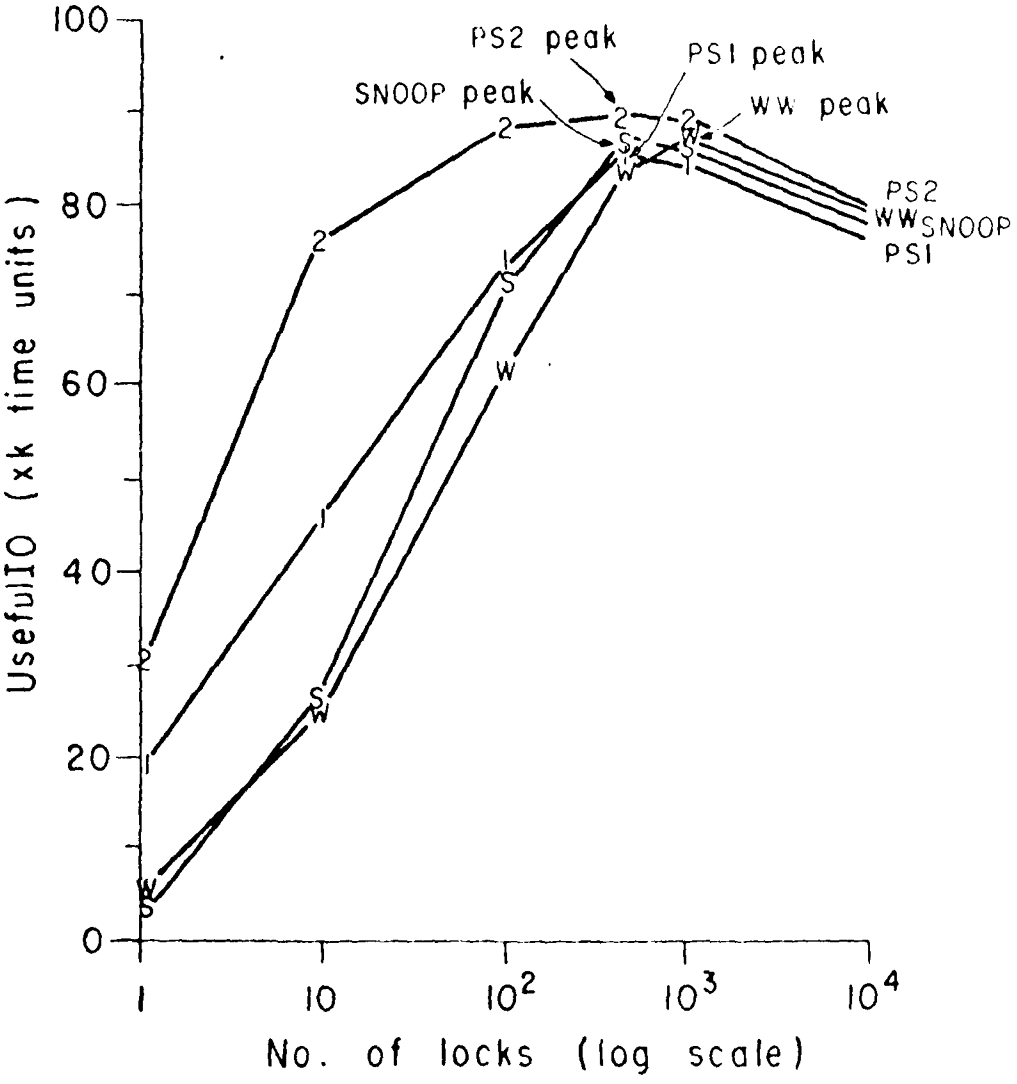


Figure 3-4: Productive Computer Utilization under Different Algorithms and Class 1 Transactions.

maximum USEFULIO was reached with 500 or 1000 granules.

Several observations about figure 3-4 should be noted. First, the primary site two model (PS2) achieved 98% of the maximum USEFULIO with 100 granules and 90% of that maximum with as few as 50 granules. Each of the other three models required at least 250 granules to reach within 10 percent of its respective maximum. Thus, more coarse granularity was acceptable in the primary site two model. In that model, no transactions held locks at one node while waiting for locks at another node. In each of the other models this condition was not true.

Second, the differences in useful computer utilizations were very small at the optimum granularities, although the primary site two model (PS2) did show a slight advantage. At lower granularities, the primary site models produced significantly more useful computer utilization since transactions did not have to be restarted. Similarly, at lower granularities, the SNOOP model out-performed the wound-wait model, since it caused even fewer transactions to be restarted.

The average response time curves (not shown) for the transactions in class 1, did not consistently favor any of the four algorithms. However, at or near the optimum granularity (1000 locks at each node), the decentralized algorithms had a better average response time than the primary site 1 model. This result is expected, since local transactions can be run without network delay.

Surprisingly, however, the average response time was even less for the primary site 2 model. In this case, any gains observed by the local transactions in the decentralized models were more than offset by the extra delay experienced and caused by distributed transactions holding locks at one node while waiting for locks at another node!

The exact values of the output parameters observed with 500 locks at each node for class 1 transactions are reported in Table 3-7.

Several observations should be noted. At the primary site 2 model, the number of transactions completed, TRAN-

Table 3-7
Output Measurements for Class 1 Type Transactions

Measurement	PS1	PS2	WW	SNOOP
TRANCOM	2,688	3,307	3,094	3,029
AVERRES	392	350	368	362
USEFULIO	86,056	87,648	87,518	87,556
USEFULCPU	86,065	86,335	87,522	87,563
LOCKCPU	843	1,048	952	962
MESSCPU	207	265	108	103
TMESS	9,697	12,437	5,408	5,178
LMESS	6,799	8,329	1,545	1,664
NWOUNDED	-	-	6	-
NCONFLICTS	-	-	-	68
NRESTARTED	-	-	1	0
DLOSTIO	-	-	392	0.0
DLOSTCPU	-	-	392	0.0

COM, was 10% greater than with the two distributed control models and about 18% greater than with the primary site 1 model. However, the differences in USEFULIO and USEFULCPU were not significantly different for the four concurrency control models. Thus, the large TRANCOM value was due primarily to the fact that the PS2 model favored smaller transactions and 90% of the workload included those small transactions.

With the other models, larger distributed transactions could block both large and small transactions at several nodes while waiting for locks at another node. With the PS2 model, however, the larger distributed transactions (which have the greatest probability of conflict), would release the locks at lower numbered nodes.

The LOCKCPU, MESSCPU, TMESS and LMESS parameters were also greater for the primary site 2 model, since more transactions had been completed.

As expected, message CPU overhead was lowest for the decentralized concurrency control algorithms. Also note that the ratio of the total number of lock messages sent to the total number of messages sent (LMESS/TMESS) is about for the primary site models versus .3 for the decentralized control models. In other words, two-thirds of the network traffic was due to concurrency control in the primary site models. Less than one-third of the messages in the decentralized models were for concurrency control.

The expected number of messages shown in Table 3-6 can be applied to the number of observed messages shown in Table 3-7 to determine the exact number of lock messages sent by local transactions. In the primary site 2 model, for example, 4,108 non-lock messages (TMESS-LMESS) had been sent. Since the expected number of non-lock messages is 12.4, 311 of the 3,307 transactions were non-local. (Note that this number is consistent with 3,307 total transactions and a PREDIST value of 10%.) Thus 2,996 transactions were entirely local and yet were responsible for 7,490 lock messages.

Notice that a very small number of transactions were wounded. In the canonical scenario only 10% of the transactions were distrbuted and only conflicts between distributed transactions could cause wounds. Furthermore, all locks are requested at the beginning of a transaction and were generally granted. Thus, a transaction is much more likely to be blocked by an older transaction, in which case no wound is sent. Note that many more conflicts than wounds were sent. However, no deadlock was detected, so no transactions were restarted in the SNOOP model.

The number of conflicts in the SNOOP model was always greater or equal to the number of transactions wounded in the wound-wait model, since all conflicts between distributed transactions were sent to the SNOOP. However, the number of killed or restarted transactions in the SNOOP model was always less than or equal to the number restarted in the wound-wait model, since only actual deadlocks could cause a restart. In fact, in the simulation results reported in Table 3-7, no transactions were restarted in the SNOOP model.

4.1.2. Class 2 Transactions

The USEFULIO computer utilization for each of the four concurrency control algorithms for class 2 transactions are shown in Figure 3-5. Under the randomly placed locks with only small transactions, the finest granularity, 10,000 locks in this case, was again optimal. With this optimal granularity, as with class 1 transactions, only slight differences in computer utilizations were due to the concurrency control algorithms.

However, the wound-wait and global deadlock detector algorithms did consistently produce somewhat better results than the primary site algorithms over a wide variety of granularities. In fact, only with fewer than 50 locks at each node, were the primary site models

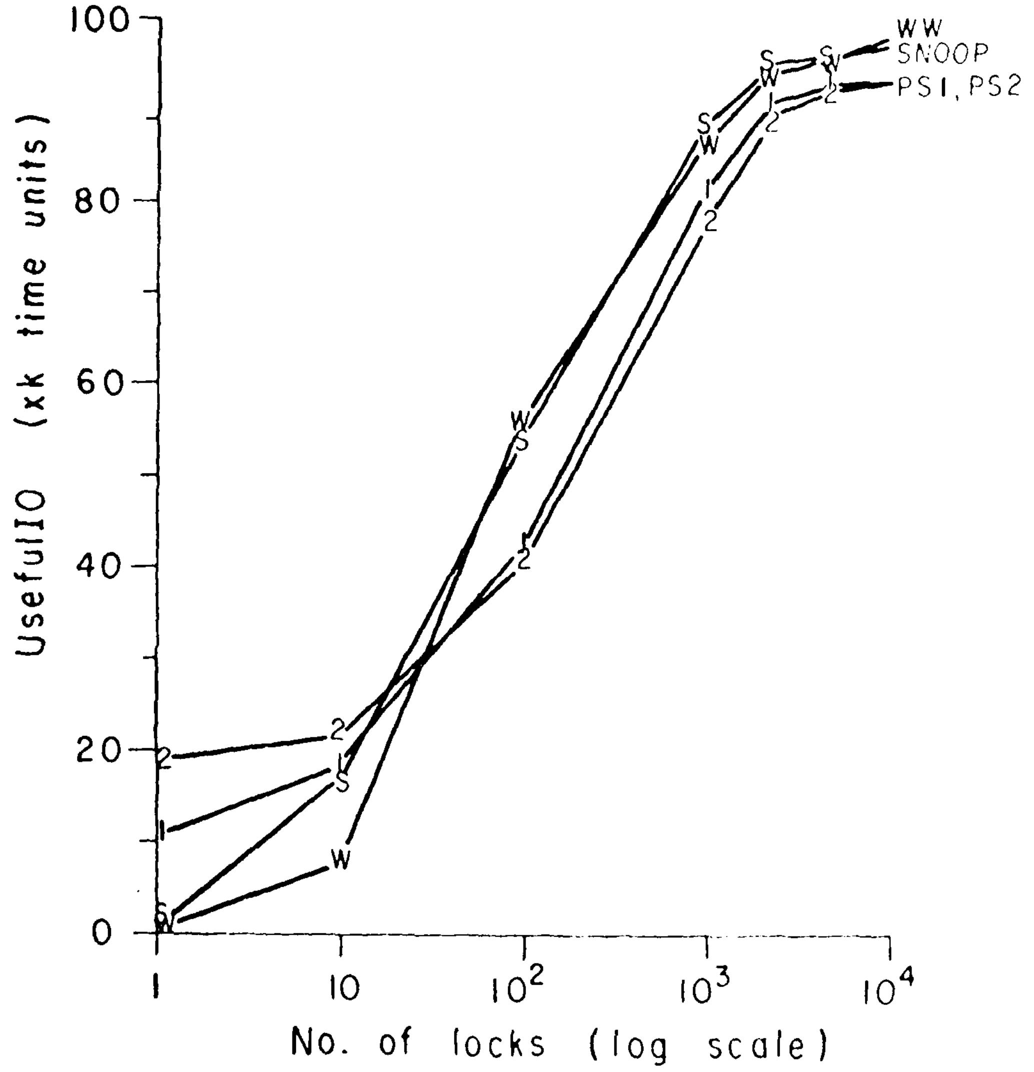


Figure 3-5: Productive computer utilization with four algorithms and Class 2 Transactions.

advantageous.

No difference in computer utilization was observed between the two primary site models once the granularity became fine enough. This result was true for class 2 transactions, since the probability of success on a lock request was extremely high. Thus, very few of these transactions waited for locks at one node, while holding locks at another node.

Similarly, once the granularity was less coarse (about 50 granules), little difference in computer utilization is realized between the two decentralized algorithms. This result was also realized because of the high probability of success on a lock request.

Figure 3-6 shows the average response time versus the number of locks at each node for class 2 transactions. The response time is given in terms of time units of the simulation. In the canonical interpretation of the time parameter, a response time of 61 would represent about 1.8 seconds. The dichotomy between the primary site algorithms and decentralized algorithms was again realized in these curves. As expected, the decentralized algorithms produced lower average response times, since local transactions did not need to communicate with any other nodes.

The exact values of the output measurements for 10,000 locks at each node are reported in Table 3-8 for

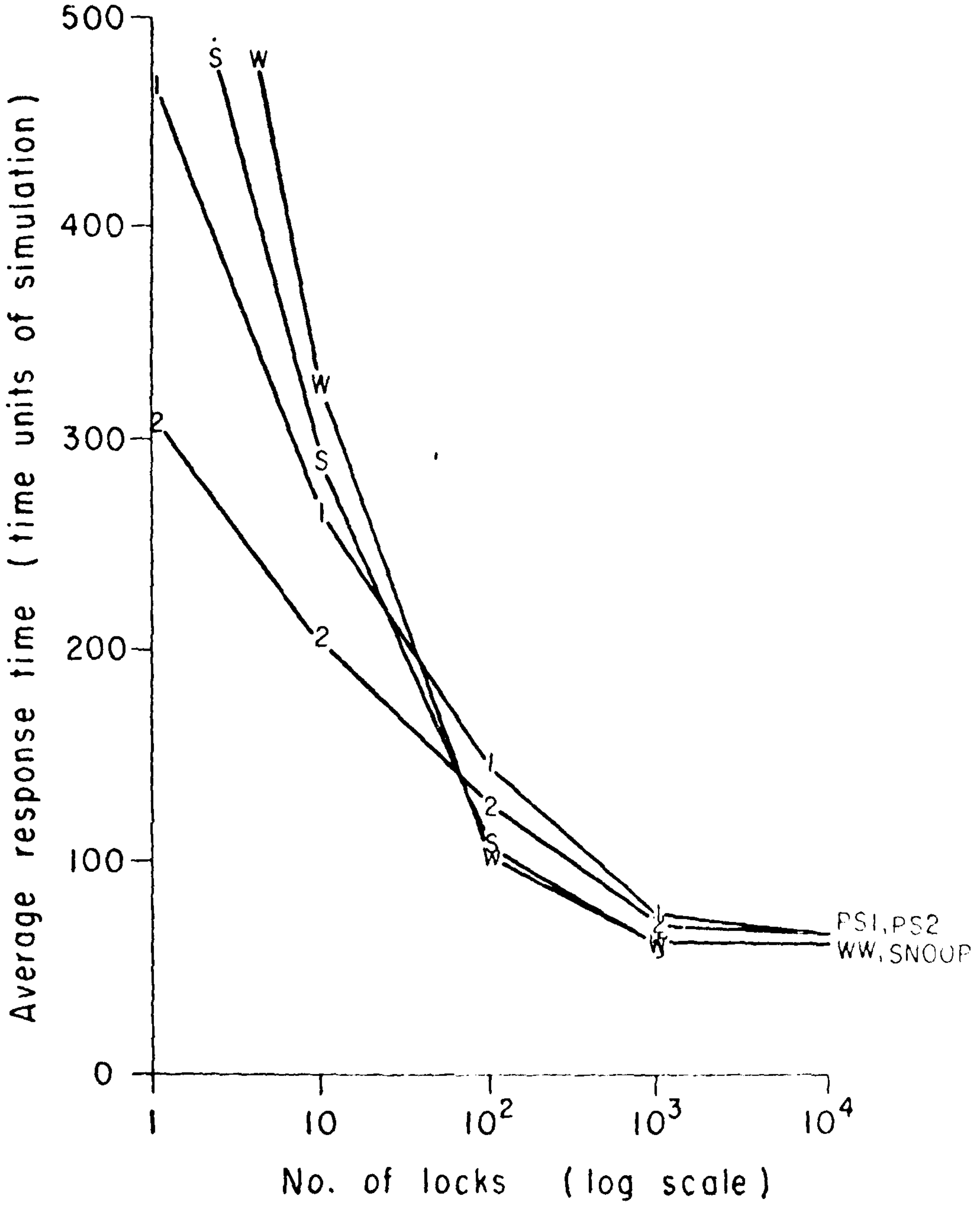


Figure 3-6: Average Response time for four algorithms and Class 2 Transactions.

class 2 transactions.

Note that the network parameters observed the same ratios of total messages to lock messages as with the class 1 transactions. However, the differences in the number of lock messages between the primary site models and the decentralized models was over 40,000 messages with class 2 transactions. With that number of messages it is no longer realistic to assume that the network is 'lightly loaded' i.e., that the message bandwidth parameter is infinite. Restricting the message bandwidth can only increase the differences between the primary site control and decentralized control models as will be shown in section 4.5.

Table 3-8
Output Measurements for Class 2 Type Transactions

Measurement	PS1	PS2	WW	SNOOP
TRANCOM	18,455	18,461	19,259	19,097
AVERRES	64	64	61	62
USEFULIO	93,956	93,280	97,135	96,193
USEFULCPU	93,996	93,319	97,145	92,204
MESSCPU	1,519	1,529	670	700
TMESS	73,977	74,292	33,520	34,852
LMESS	51,594	51,234	9,635	11,093
NWOUNDED	-	-	-	-
NCONFLICTS	0	-	-	5
NRESTARTED	-	-	0	0
DLOSTIO	-	-	0	0
DLOSTCPU	-	-	0	0

The relatively small differences in average response times between the primary site and decentralized control models, was at first surprising. However, most of the delay for the transactions was due to competition for the CPU and I/O resources. The network delay time of 2 * MES-RATE time units was not a relevant factor. For example, with ten transactions at each node, a local transaction was active at a site with 9 other transactions. For class 2 transactions, the average transaction size was 5. Thus a transaction waited for the I/O and CPU resources for about 45 time units (9 trans X 5 time units/trans). In addition, the average transaction would spend 5 time units using the I/O and CPU resources.

Thus 55 time units of the average response time is accounted for without considering lock conflicts or net-work delays. If either fewer transactions were running, the transaction sizes were smaller, or the network were slower, the 2 * MESRATE delay would further increase the response time difference between the primary site and decentralized models.

The expected number of messages shown in Table 3-6 can also be used in analyzing the number of messages shown in Table 3-8. In this case, 1,859 transactions were non-local in the primary site 2 model. Thus, the 16,602 local transactions accounted for 41,505 of the lock messages.

Note, however, that the number of lock messages in both primary site models is higher than the expected number of messages according to Table 3-6. For 18,461 transactions completed, 46,153 (TRANCOM * 3(5/6)) lock messages should have been sent. This difference was due to a slight bottleneck at the primary site. In computing the expected value it was assumed that 5 out of every 6 transactions completed would be initiated at other sites and thus require the lock messages. However, due to the bottleneck at the primary site, 9 out of every 10 transactions were initiated at other sites.

In the next sections, the effects of variation in the input parameters on the above observations are reported.

4.2. Slave Transactions

In a distributed database, not all of the distributed transactions require access to data at all of the nodes as assumed in the above results. In this set of experiments, the number of SLAVES required by each MASTER transaction, NSLAVE, was set to 1, 3 and 5. With these settings of NSLAVES, a distributed transaction thus accessed data at 2, 4 and 6 nodes, respectively. The results of these parameter settings for class 1 and class 2 type transactions for the four concurrency algorithms follow.

4.2.1. Class 1 Transactions

The effects of varying the number of SLAVES were similar under any of the four concurrency control algorithms. The maximum useful computer utilization again occurred with 500 or 1000 granules regardless of the number of SLAVES used by a distributed transaction. In addition, all four concurrency control algorithms resulted in similar shifts in the utilization curves as the number of nodes per distributed transaction varied. The shifts are shown for the SNOOP algorithm in Figure 3-7.

As expected, as the number of SLAVES decreased, the useful computer utilization increased. Although the optimal granularity did not change, the number of granules required to achieve utilization close to the maximum decreased as the number of SLAVES decreased. If each MASTER transaction had 1 SLAVE, 50 locks resulted in 96% of the computer utilization realized with 500 locks. With 3 SLAVES, 91% of the maximum utilization was realized with 50 locks, while only 63% was realized if there were 5 SLAVES for each MASTER transaction. Thus, as the number of remote nodes decreased, the acceptable granularity results resembled those observed in Chapter 2 for the centralized database.

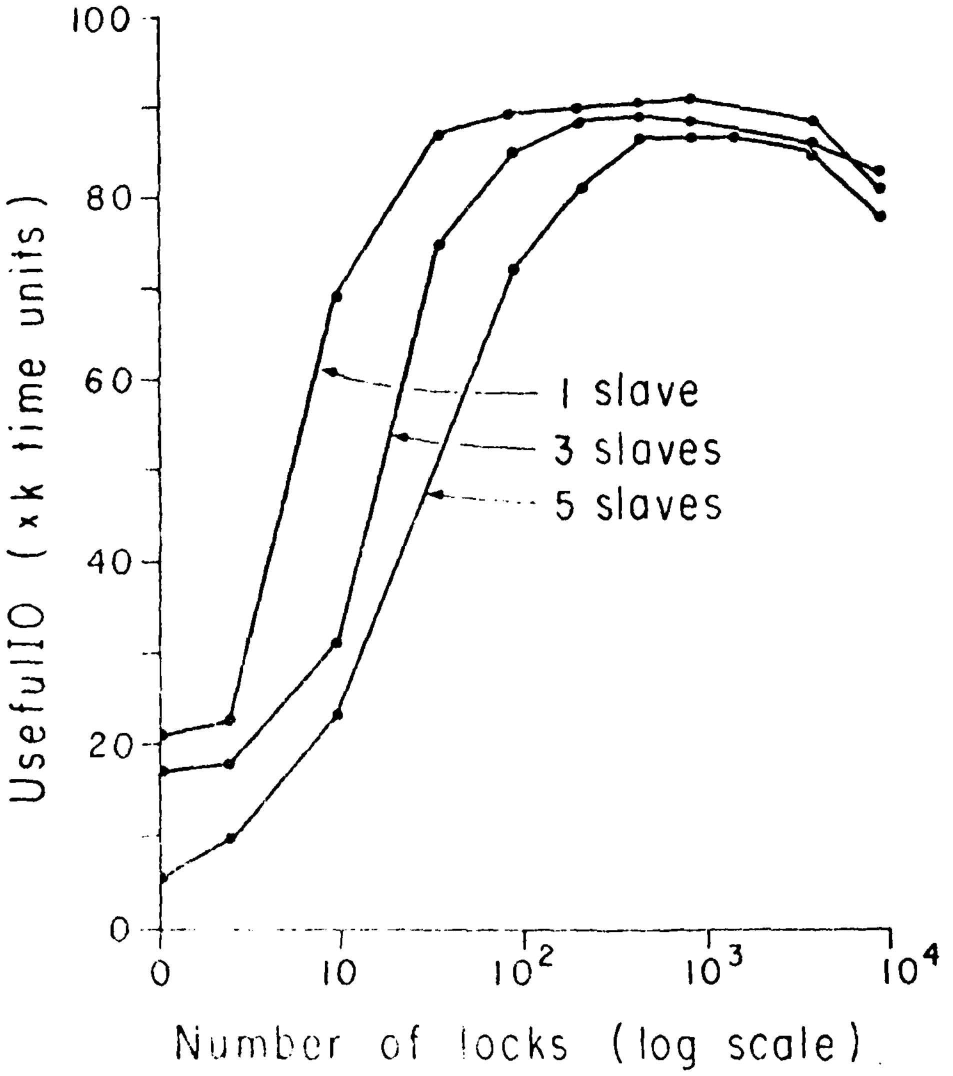


Figure 3.7: Productive computer utilization with different number of slaves and Class 1 Transactions.

As previously stated, the other three models behaved similarly. In general, varying the number of SLAVES for distributed transactions did not have a large impact on the processing at the nodes. However, the utilization of the network as a function of the number of remote nodes does depend on the concurrency control algorithm used. Table 3-9 shows the percentage of "useful" messages (non-lock related) for each of the four algorithms. With the primary site models, the number of lock messages stayed constant, but the number of non-lock messages depended on the number of SLAVES for each MASTER transaction. With the decentralized algorithms, of course, the number of lock messages decreased as the number of slaves decreased.

4.2.2. Class 2 Transactions

The number of SLAVES for a distributed transaction also had little effect on the choice of concurrency control algorithm or granularity for class 2 transactions. As in the canonical scenario, the finest granularity was again optimal. In the primary site models, the [illegible]

Table 3-9 Useful Network Traffic
(Non-lock Messages)

No. of SLAVES	PS1	PS2	WW	SNOOP
1	9%	11%	71%	66%
3	21%	21%	71%	66%
5	30%	33%	71%	68%

utilization and average response time as a function of the number of locks were almost identical for one, three and five SLAVES for each MASTER transaction. For the decentralized concurrency control models, those three curves were nearly identical with more than 50 granules. With fewer granules, more SLAVES resulted in more transactions being restarted. In these cases, the computer utilization was decreased. However, even with only one slave per distributed transaction, performance of the system with class 2 transactions was still extremely bad with coarse granularity.

The observations on the network utilization for class 1 transactions also hold for transactions in class 2. In fact, while the number of total and lock messages changed, the percentages of useful messages were approximately the same.

4.3. Number of Network Nodes

The number of sites in a distributed database can vary. The simulation models were run with 2, 4, 6 and 8 sites for a variety of granularities. In order to keep the other factors constant, the canonical scenarios were changed. In all of these experiments, it was assumed that each distributed transaction required only one slave running at another site.

4.3.1. Class 1 Transactions

With mixed transaction sizes and well-placed locks, there was practically no difference between the four concurrency control algorithms as the number of nodes in the network varied. Moreover, neither the optimum granularities nor the shapes of the useful utilization versus granularity curves changed as the number of nodes in the network varied. The curves all resembled those shown in figure 3-4.

The only changes in the computer utilizations were in magnitude, and those changes were linear with respect to the number of nodes. Note, however, that it is also assumed that the network resources also increase as the number of nodes increase. Under the wound-wait simulation, for example, with 2 nodes the maximum useful utilization was 30,119 time units; with 4 nodes, 59,705 time units, with 6 nodes, 90,472 time units, while with 8 nodes, 120,327 time units were used in processing transactions.

The average response time, on the other hand, did not vary as the number of nodes changed.

4.3.2. Class 2 Transactions

Linearity in computer utilization as a function of the number of nodes was also observed for class 2 type transactions. The USEFULIO's, USEFULCPU's and average response time for the decentralized concurrency control algorithms were slightly better than those measurements for the primary site with 2, 4, 6 or 8 nodes in the computer network.

The cost of locking with the many small transactions and the random placement of locks assumption, is, of course, much greater than with the class 1 transactions. This cost also increased linearly with the number of nodes and was practically the same for all four algorithms at the optimum granularity. The lock costs for the primary site 1 model are shown in Table 3-10:

The time units per node remained relatively constant. However, for the decentralized concurrency control algorithms, the time units used for locking were distributed among all of the nodes. In the primary site models all of

Table 3-10: Time Units Spent Locking

No. of Nodes	Total Time Units	Time Units per Node
2	3608	1704
4	6763	1691
6	10010	1673
8	13300	1663

the time units were used for locking at one node. Thus, at the primary site with 8 nodes in the network, 13,300 out of 20,000 available time units were used for locking.

This increasing overhead for locking at one node has two implications. First, transactions which use the primary site for data access will receive much poorer service than the other nodes. In fact, it may be necessary to reduce the transaction processing load at the primary site node. Second, the primary site can become saturated just managing locks. With class 2 transactions and the locking overhead rate assumed in these experiments, an extrapolation shows that the primary site will saturate if there are 12 nodes in the network. Note that the primary site also has to handle a disproportionate share of the messages. The time units used for handling lock messages (MESCPU) at the primary site should also be included in looking at primary site saturation. An extrapolation of the total overhead (LOCKCPU + MESCPU) shows that the primary site would saturate with only 11 nodes in the network.

For the class 1 transactions, on the other hand, each transaction required much less locking overhead due to the well-placed lock assumption. Under those assumptions, the primary site would not bottleneck until 83 nodes were in the network.

4.4. Percent of Distributed Transactions

In the previous simulation runs, ten percent of the transactions were assumed to be distributed, while the other transactions required processing at the local nodes. In this section, the effects of varying that percentage on the optimum granularity and choice of concurrency control algorithms are examined. Experiments were run with values of 0, 10, 25, 50, 75, and 100 for the percentage of distributed transactions parameter (PREDIST). The results are presented for both class 1 and class 2 transactions.

4.4.1. Class 1 Transactions

Changes in the percentage of distributed class 1 transactions affected the optimum granularities differently for the different concurrency control algorithms. In addition, as that percentage increased, the choice of a 'best' algorithm for class 1 transactions became clearer.

The results of the simulation experiments, varying the PREDIST parameter, are broken into the following four parts. First the effects of the locking granularities on the four models are discussed. Next the four models are compared, choosing the optimal granularity for each model for each setting of the PREDIST parameter. Third, the four models are compared under alternate network assumptions. In the final set of experiments, some messages useful in terms of crash recovery were added to the primary site model.

4.4.1.1. Effects of Locking Granularity

With any of the four concurrency control algorithms, if 0% of the transactions were distributed (all transactions are local), the maximum useful computer utilization occurred with from 50 to 500 lockable granules. These results were similar to the centralized database case in Chapter 2.

The optimum locking granularity for three of the four concurrency algorithms changed as the percentage of distributed transactions increased. With the primary site 2 model, however, the maximum useful computer utilization occurred at or near 500 granules.

For example, in figure 3-8, the shapes of the useful I/O curves versus the number of locks are very similar when either 10% or 75% of the transactions are distributed. For the other three models, 75% distributed transaction curves were skewed to the right when compared to the 10% curves.

The difference between the models is that in the primary site 2 model, no transactions hold locks at one node

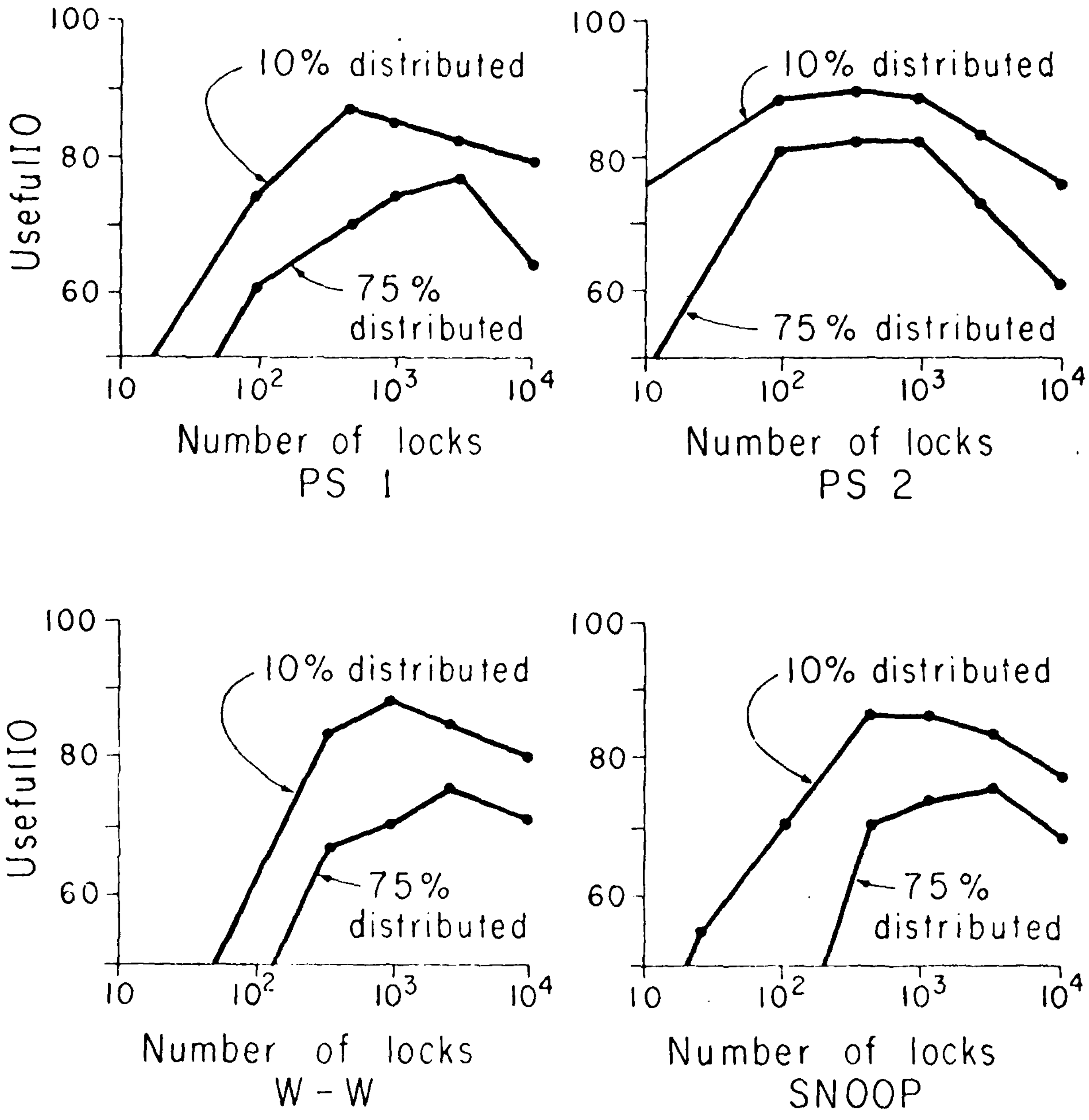


Figure 3-8: Effects on Productive Computer Utilization of Locking Granularity and Percent Distributed

while waiting for locks at another node. As the percentage of distributed transactions increased, there was an increase in the number of transactions which held locks at the other nodes in the other models. Lower granularity increased the number of incidences of this condition and hence adversely affected the performance of those algorithms.

The effects of varying the granularity and the percentage of distributed transactions on the decentralized algorithms was even more dramatic. For these algorithms, a granularity from 1000 up to 5000 locks at each node was required to produce the maximum computer utilization as the percentage of distributed transactions increased beyond 50%.

The need for finer granularity in these cases was caused by two effects. First, as already mentioned, transactions hold locks at one node while waiting for locks at a second node. The second factor affecting the granularity in these models was that with coarse granularity and a high percentage of distributed transactions, more transactions had to be restarted.

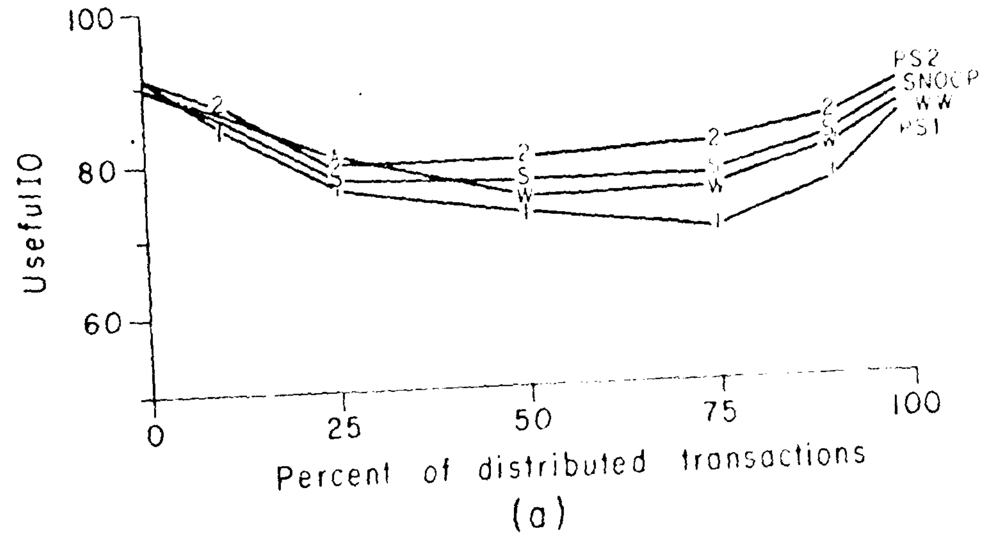
4.4.1.2. Model Comparisons

Figure 3-9 shows the effects on the useful I/O and the average response time of the percent of distributed transactions for each of the four concurrency control algorithms. (For each percentage, and for each algorithm, the best useful I/O and average response time regardless of granularity was plotted.)

The 'dish' shaped curves for USEFULIO were surprising. As the percentage of distributed transactions was increased up to 50%, all four models showed decreases in useful computer utilization due to the additional overhead (message handling and locking) required to run distributed transactions. However, as the percentage increased beyond 75%, the useful computer utilization significantly increased.

That increase was due to two factors. First, the number transactions running at each node was greatly increased. For example, when all of the transactions were distributed, NNODES * NTRAN (60 in the simulation runs) parts of transactions were active at each node. Second, the average transaction size at each node was smaller as more and more transactions were distributed.

The simulation parameters were modified to keep the number and sizes of active transactions at each node con-



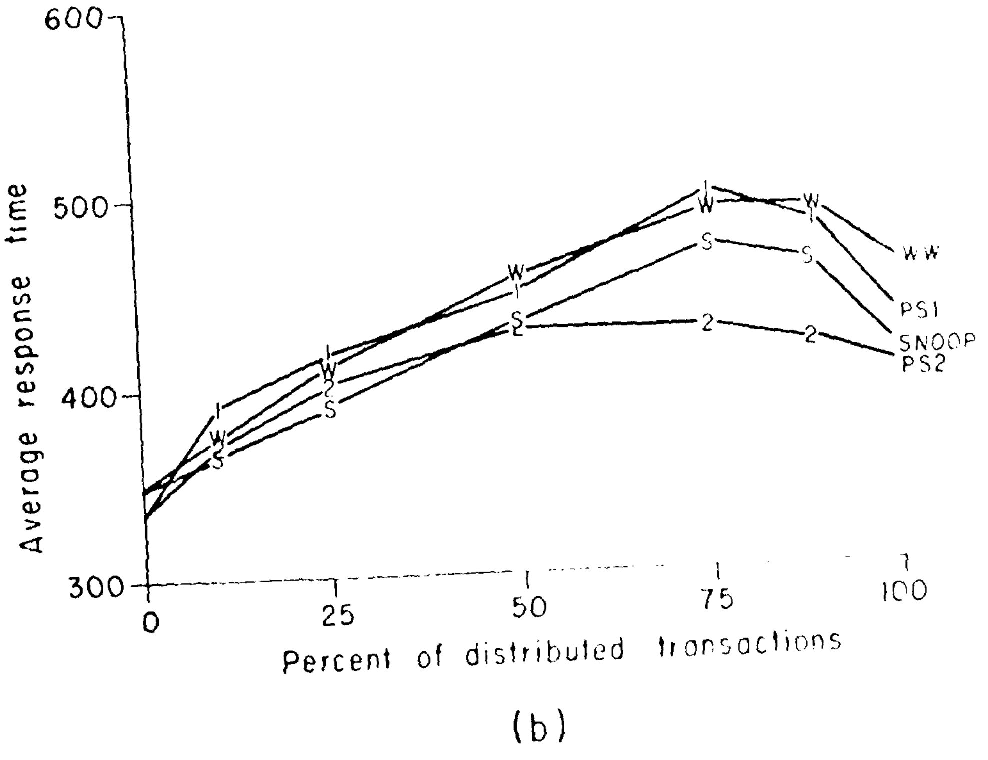


Figure [illegible]: Class 1 Transactions
Infinite Bandwidth

stant as the percentage of distributed transactions increased. Only when both parameters were held fixed did the 'dish' shaped curves disappear. When only one of the parameters (NTRAN or AMEAN-BMEAN) were held constant, having all transactions distributed produced more useful I/O (and CPU) than when only 50% of the transactions were distributed.

The average response time curves also demonstrated dish shaped curves. In almost all cases, the second primary site model (PS2), produced the best average response time of the four models. The holding of locks at one node while waiting for locks at another was quite detrimental to the throughput of the system and occurred with increasing frequency in the other three models as the percentage of distributed transactions increased.

When fewer than half of the transactions were non-local the SNOOP and PS2 models produced about equal useful I/O and average response times and were slightly better than the other two models. However, when more than half of the transactions were non-local, the primary site 2 model produced significantly better results than the other three models.

4.4.1.3. Limited Bandwidth

The above observations change if a lower network bandwidth was assumed. All four concurrency control simulations were rerun, varying the percentage of distributed transactions with a message bandwidth of 6. This simulates an environment where only six messages can be active in the Network one at a time. The tests included locking granularities of 500, 1000, 2500 and 5000 locks at each node. Additional values for the PERDIST parameter were also tested and included 30, 35, 40 and 45 percent. The results are shown in Figure 3-10.

With fewer than 40% of the transactions being non-local, the global deadlock detector algorithm produced more useful I/O utilization than the other algorithms. When 45% or more of the transactions were distributed, the primary site 2 model again produced better results. In these cases, the extra two messages for locking were not that significant; a distributed transaction required at least 2 * NSLAVES messages anyway.

Note also that the 'dish' shape curves for USEFULIO have practically disappeared with a limited bandwidth network. In these cases the extra network delay overhead caused by an increased PREDIST parameter more than offset the increases in transaction parallelism.

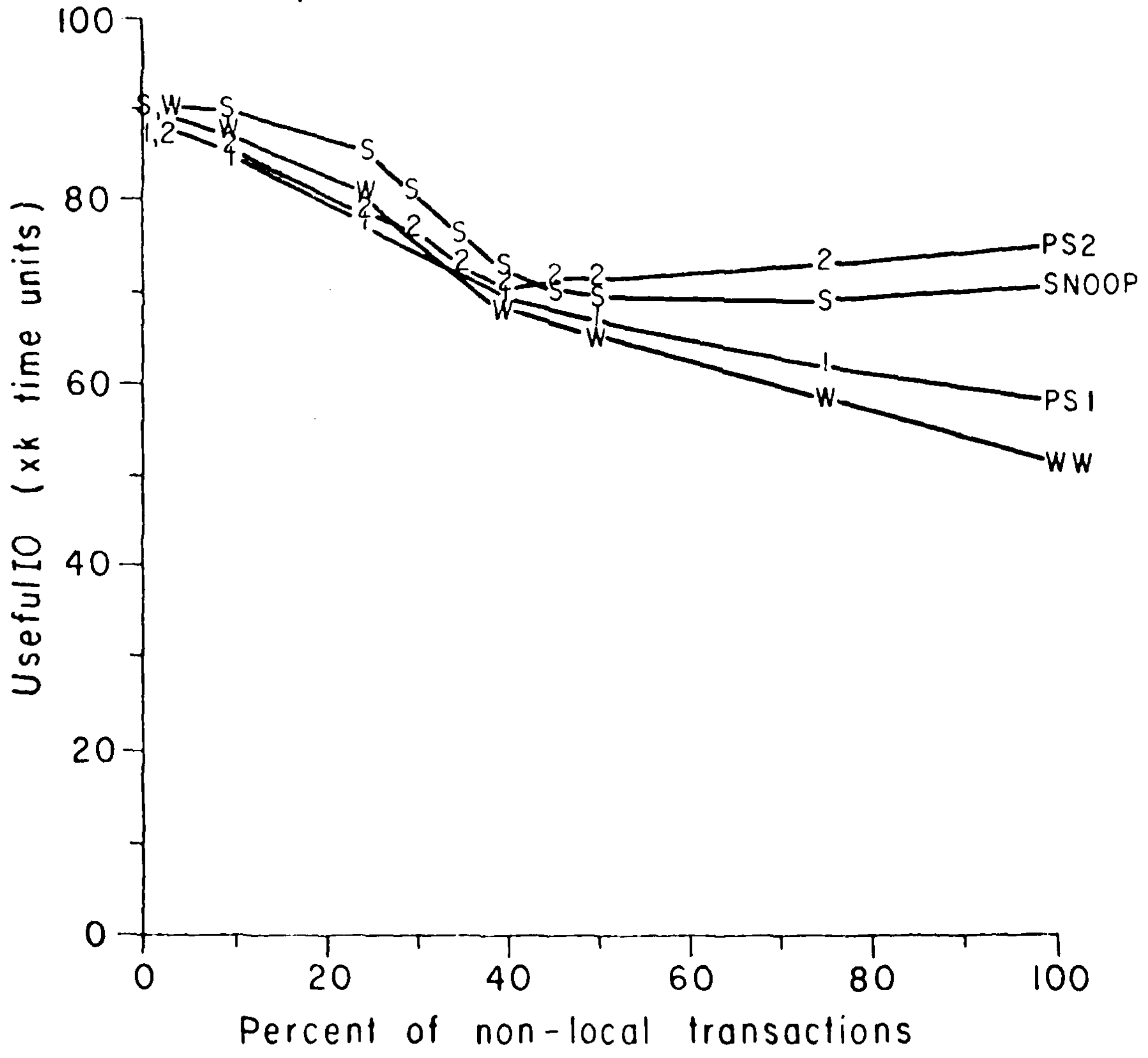


Figure 3-10: Class 1 Transactions
Limited Bandwidth

4.4.1.4. Alternate Primary Site Model

Those differences between the SNOOP and PS2 models would be even less, if the primary site models required the 'release lock' messages to be sent to the SLAVES. In many database management systems, transactions might be backed out due to system crashes, changes in a user's mind and a variety of other reasons. For these reasons, it may be desirable to have SLAVES wait until the transaction has completed at all nodes before 'committing' any updates. In these types of database management systems, 'all done' messages similar to the 'release locks' messages must be sent to the SLAVES even with the primary site concurrency control.

The primary site 2 model was modified to actually send "all done" messages at the end of each distributed transaction. With that modification and the limited bandwidth network, the primary site 2 model actually produced slightly less useful computer utilization than the SNOOP model, regardless of the percentage of distributed transactions.

4.4.2. Class 2 Transactions

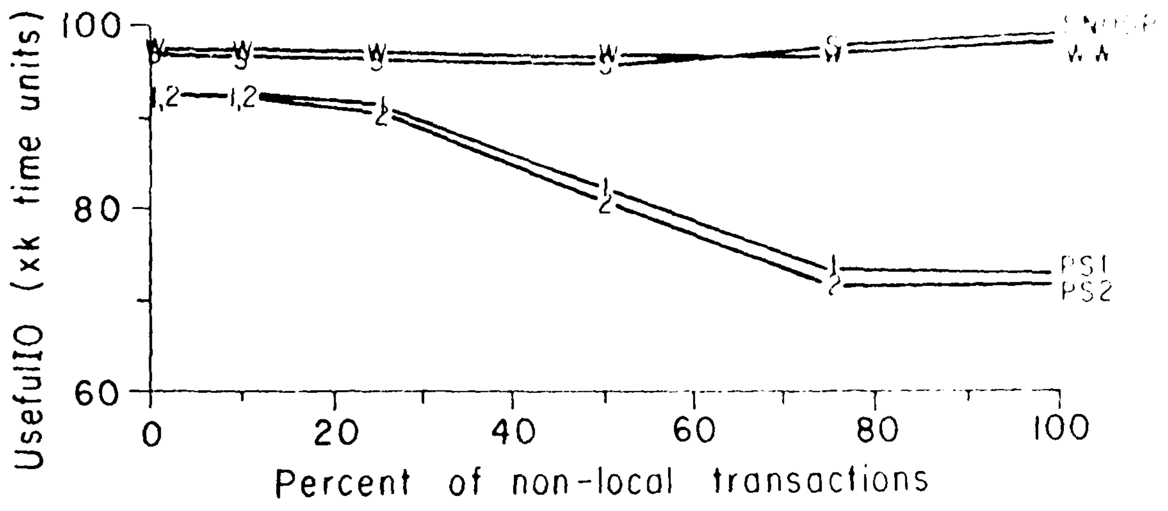
With class 2 transactions, the finest granularity was optimal, regardless of the percentage of distributed transactions. Furthermore, the performance of the concurrency control algorithms also changed consistently as the percentage of distributed transactions increased.

Figure 3-11(a) shows the USEFULIO for the four algorithms as that percentage increased. The utilization with the decentralized algorithms was affected very little by the increase in non-local transactions. Again, a slight increase in useful computer utilization was realized due to the increased distribution of transaction processing.

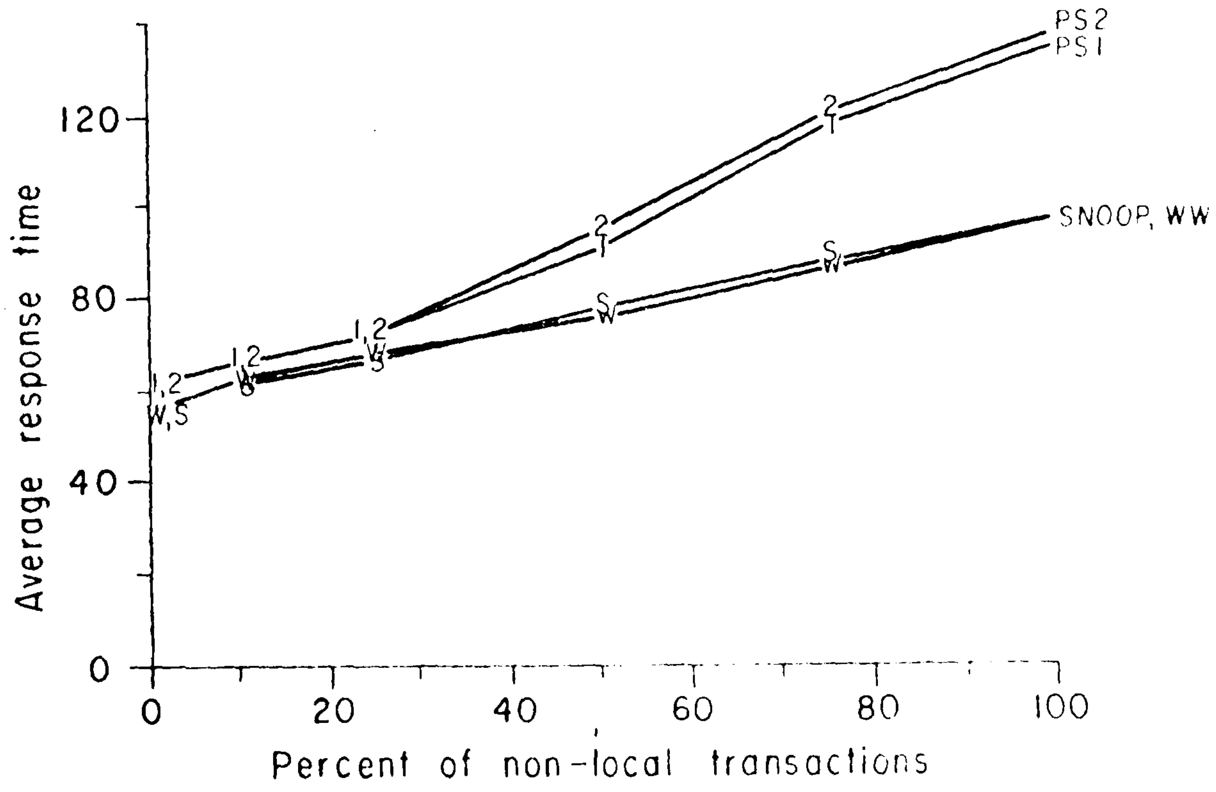
In the primary site algorithms, on the other hand, the overall computer utilization decreased as the percentage of non-local transactions increased. The decrease was most dramatic between 25 and 75 percent.

The same advantage for the decentralized algorithms over the primary site algorithm appeared in the average response time, as shown in figure 3-11(b). For all four algorithms the response times increase as the percentage of distributed transactions increased. However, the increase was much less for the decentralized concurrency control algorithms than for the primary site concurrency control algorithms.

Two factors caused the dramatic difference between the primary site and decentralized models for class 2 transactions: the transactions were all small and the primary site created a bottleneck.



(a)



(b)

Figure 3-11: Class 2 Transactions
Infinite bandwidth

The transactions of class 2 were all small and the results in Figure 3-11 were for the finest granularity. Under those conditions, the probability of success on a lock request was extremely high, which considerably reduced the advantage that the primary site 2 model exhibited for class 1 type transactions.

The second factor which affected the performance of the concurrency control algorithms was the bottleneck at the primary site. Over 7,000 time units out of a possible 20,000 were used for locking at the primary site when all of the transactions were non-local. Moreover, all transactions required some database processing at that primary site and were thus all delayed by the locking overhead. This bottleneck became increasingly worse as the percentage of distributed transactions increased.

One solution to the bottleneck problem would be to offload the primary site concurrency control to a separate processor. The primary site 2 simulation was modified to test this strategy.

Two sets of experiments were run. In the first set, the workload and network parameters remained the same and the concurrency control was off-loaded to a 'seventh' node. In these experiments, the primary site model produced USEFULIO and average response times very similar to the decentralized control algorithm results shown in figure 3-11. In fact, the primary site models produced slightly better results than the decentralized models when the PREDIST parameter was greater than 50%.

In the second set of experiments, the 6-node database, granules and transactions were distributed on a 5-node network with a sixth node being used only for the concurrency control. The results were again similar to those in figure 3-11 for the decentralized models. However, in these experiments the modified primary site models produced slightly worse results than the decentralized models.

These two results suggest that a proper database design which lowered the load at the primary site could perform equally as well as the decentralized algorithms.

The PREDIST simulation experiments for class 2 transactions were repeated with a limited bandwidth network. In these experiments, the primary site models were best if more than 50% of the transactions were distributed. In those cases, the primary site models actually sent fewer locking messages than the decentralized algorithms.

4.5. Network Parameters

In this section, the results of varying five network input parameters are reported. In the previous runs the

MESRATE, or the length of time it takes to send a message, was fixed at 3 simulation time units. The MESBDWT, or number of simultaneously active messages, was effectively set to ∞, by setting the MESBDWT parameter to 1000.

The data transfer parameters, PRETRAN and PREDATT, were also fixed in all of the previous simulation experiments. In those experiments 40% (PRETRAN) of the distributed transactions sent 25% (PREDATT) of their entities to other nodes. The DATARATE parameter was set to .05, which determined how long it took to send data entities across the network.

One other network parameter, the MESSCPURATE, while not affecting the network directly, did affect the message or network overhead required at each node. For all of the previous experiments, a message CPU rate of .01 (300 microseconds) was assumed.

Simulations were run with MESRATES of 1 (30 msecs), 3 (90 msecs) and 10 (300 msecs, similar to the ARPANET). The simulations were also run with MESBDWT of 100, 50, 10 and 6. The DATARATE experiments included, 0.05, 0.1, 0.25 and 0.5. The message CPU rate parameter was set to .01 (300 microseconds), 0.05 (1.5 msecs), 0.1 (3 msecs) and 0.3 (9 msecs).

Class 2 transactions required much greater use of the network resources than class 1 transactions. Thus variations in the network parameters had a much greater effect on class 2 transactions.

4.5.1. Class 1 Transactions

The significant effects of lowering the bandwidth and varying the percentage of distributed transactions have already been reported in section 4.5. Varying the MES-RATE, MESBDWT and MESCPURATE parameters had little effect on the other observations reported.

The effects of varying the message rate parameter were slight. The results with message rates of 1 and 3 were almost identical for all four concurrency control algorithms. A MESRATE of 10 resulted in about a 5% decrease in useful computer utilization for the primary site models and almost no change in the useful utilization for the distributed concurrency control models.

MESBDWT settings of 100 and 50 produced useful computer utilizations and average response time identical to the infinite setting 1000 previously used. Slight drops in the useful I/O and CPU utilizations were realized with message bandwidths of 10 and 6. The drops with a message bandwidth of 10, however, were less than 1% and not considered significant.

A message bandwidth of 6 did produce more noticeable reductions in the useful I/O and CPU utilizations. The drops in useful utilization were only about 2-3% with the primary site and SNOOP models. The wound-wait model, on the other hand, realized a drop of almost 7%. Although the primary site models sent more lock messages, they were mainly sent one message at a time. A wound or kill, however, resulted in NSLAVE messages being sent, or broadcast over the network. These "bursts" of messages were effected more by the lower bandwidth than the greater number of individual messages in the primary site models. In the SNOOP model, on the other hand, a conflict only required 1 message. A kill still required NSLAVE messages, but occurred very rarely.

The change of the DATARATE parameter had little effect on class 2 transactions. When the DATARATE was .5 and all of a distribute transaction's entities were sent across the network, a decrease in the computer utilization of only about 7% was realized.

With an extremely fast DATARATE parameter (.05 as in the canonical scenarios), changes in the number of transactions which transferred data, or the amount of data they transferred produced curves almost identical to those shown in figures 3-4, 3-5 and 3-6, and are not repeated here. A slight drop in useful I/O and CPU time was observed as the amount of data transferred increased for both classes of transactions and for each of the concurrency control algorithms. However, even if all of the distributed transactions transfer all of their data, the decrease was less than 3%.

Note that these results do not imply that data transferred is not an important parameter in a distributed database. In the models considered here, data transfer resulted in a waiting time for that transfer to complete. Under these assumptions, no additional I/O or CPU resources were used in transferring data; it was assumed that use of these resources is already included in transaction processing. Furthermore, with the fast DATARATE assumed, even a transaction accessing 500 entities would wait on the transaction wait queue for only 25 time units.

When the DATARATE was increased from .05 to .5, and the PRETRAN and PREDATT parameters were varied, a larger drop in useful CPU and I/O utilization was observed. At the optimum granularity, a drop of almost 7% in computer utilization was realized. In these cases, the larger transactions might wait on the CPU queues for 250 time units, a significant portion of their lifetimes.

Changes in the MESCHURATE parameter had the greatest effect on the useful computer utilization output parameters. In the primary site models, a decrease of almost 9%

was realized when the message rate was increased to .3 (almost 9 msecs). With that same message rate, the useful computer utilization only dropped by about 4% in the decentralized models.

In class 1 transactions, the critical resources ar the I/O and CPU resources at the nodes and not the network resources. Thus the heavy message traffic of the primary site models is impacted much more by the message CPU rate than the other network parameters.

4.5.2. Class 2 Transactions

The MESRATE, MESCPURATE, MESBDWT, and DATARATE parameters were also varied for class 2 transactions. Changes in the first three parameters affected the performance of all four concurrency control algorithms. The DATARATE parameter had practically no effect on the processing of class 2 transactions.

The USEFULIC and the average response time (in parenthesis) is given in Table 3-11 for each of the four concurrency control algorithms. In the first set, the MESRATE parameter was varied while the MESCPURATE and MESBDWT were fixed at .01 and 1000 respectively. As the message rate increases, the gap between the primary site and decentralized control models widened.

AD-A103 141 CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB F/G 9/2
THE EFFECTS OF CONCURRENCY CONTROL ON DATABASE MANAGEMENT SYSTE--ETC(U)
APR 70 D R RIES N00039-78-C-0013
UNCLASSIFIED UCB/ERL-M79/20 NL

3 OF 3
AD A
103141

END
DATE
FILMED
9-81
DTIC

Table 3-11: Effects of Network Parameters

	PS1	PS2	WW	SNOOP
MESRATE				
1	94994(63)	94720(63)	96839(61)	97037(62)
3	93996(64)	93319(64)	97134(61)	96204(62)
10	87998(67)	88078(67)	96037(63)	96875(62)
MESCPURATE				
.01	93996(64)	93319(64)	97145(65)	96204(62)
.05	88953(67)	88767(68)	95048(63)	94710(64)
.1	83273(72)	83086(73)	92394(65)	91860(65)
.3	58676(102)	58372(102)	83313(72)	82690(73)
MESOULP				
1000-50	93996(64)	93319(64)	97145(61)	96204(62)
10	82804(72)	83234(72)	96827(62)	96979(62)
6	55200(108)	55692(108)	95948(63)	96242(62)

A more dramatic change occurred when the message CPU rate was varied. During these experiments, the MESRATE and MESBDWT were fixed at 3 and 1000 respectively. With a 3 millisecond cost (MESCPURATE = .1) for sending a message, the primary site models produced only 89% of the useful computer utilization that was realized with the decentralized concurrency control algorithms. With a 9 msec message rate (MESCPURATE = .3) this percentage drops to 72%.

Similarly, a dramatic change in USEFULIO and response time for the primary site models was realized as the message bandwidth was restricted. For these experiments, the message rate and message CPU rate parameters were fixed at 3 and .01 respectively. Note that while the performance of the primary site models was heavily affected by the restricted bandwidth, the decentralized models were hardly affected at all. This result is due to the fact that with the primary site models, almost 40,000 more messages were sent than with the decentralized algorithms.

Variations in the DATARATE, PRETRAN and PREDATT parameters had little or no effect on the performance of the four concurrency control algorithms. Class 2 transactions were all small. Thus any wait on the data transmission queue was also small even if all of the distributed transactions transferred all of their data.

As expected, the performance of a primary site concurrency control algorithm deteriorated as restrictions were placed on the network. The effect of the restrictions on the wound-wait and SNOOP algorithms was much smaller.

4.6. Canonical Scenario Revisited

In section 4.1, the effects of the different concurrency control algorithms on computer utilization and average response times with two different classes of transactions were presented. In those experiments a very fast, low overhead and high bandwidth network was assumed.

Subsets of those cases were rerun under alternate network assumptions. For the results presented in Figure 3-12, the MESRATE was assumed to be 10 simulation time units or about .3 seconds. The MESBPWT parameter was set to 6, while the MESCPURATE was set to .1, simulating a cost of about 3 msecs to handle a message at a node. These settings roughly resemble the ARPANET parameters. Note that the simulations were not run for all of the granularities.

In section 4.1 for class 1 transactions with finer granularities, no one concurrency control algorithm seemed dominant. Figure 3-12 shows, on the other hand, that the decentralized algorithms, the wound-wait or SNOOP, produce significantly better machine utilization than the primary site models. The drop of about 9% realized with the primary site models, when compared to the decentralized models, is consistent with the drop observed in section 4.6, when only one of the network parameters was varied.

The advantage of the decentralized algorithms for class 1 type transactions shown in section 4.1 became even more apparent when a slower network was assumed. Note, however, that under the given network parameter the useful computer utilizations for even the decentralized algorithms were much lower than with the original network parameters. Thus, regardless of the concurrency control

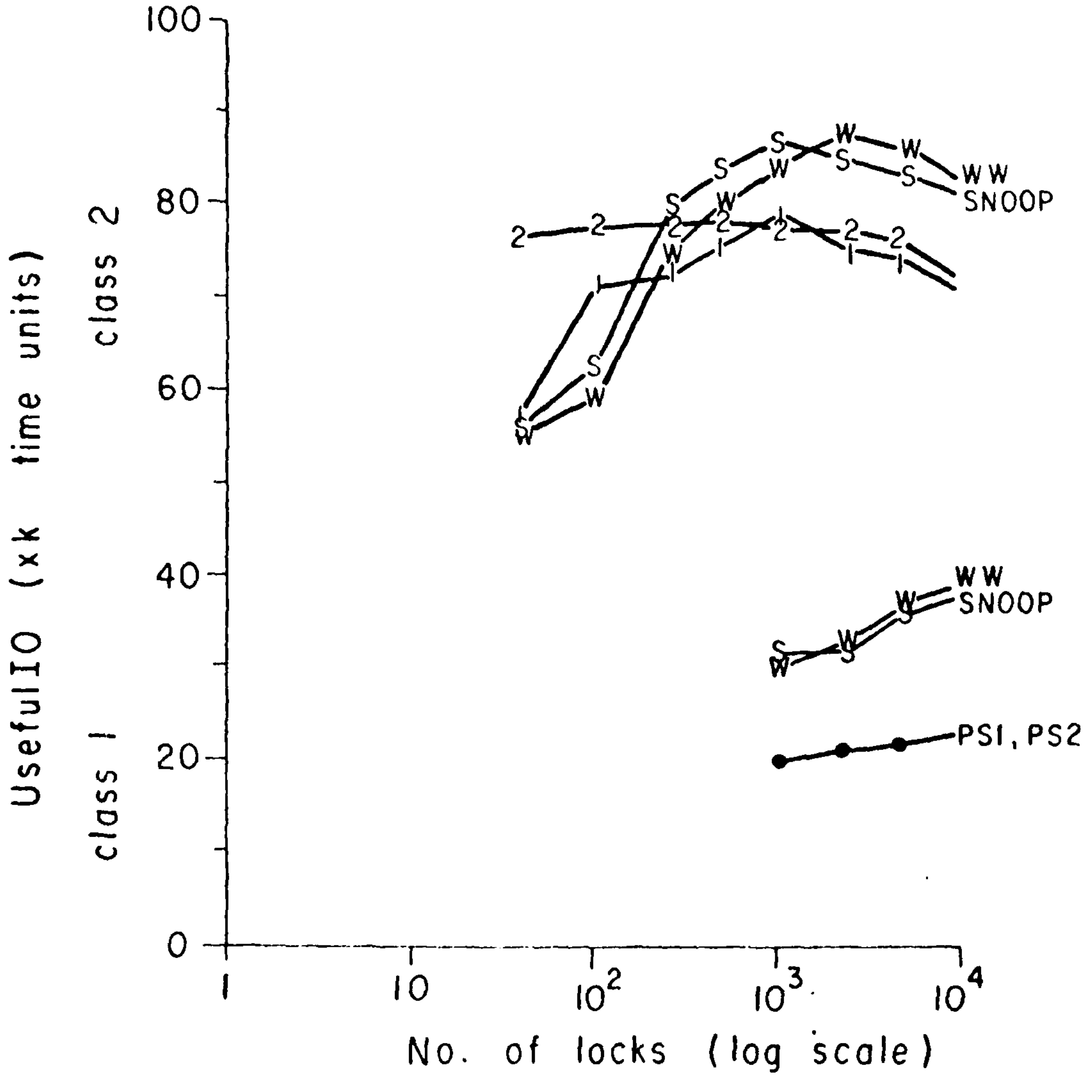


Figure 3-12: Canonical Scenario
Limited Bandwidth

algorithm, a distributed database where all transactions are very small is perhaps not suitable for a slow computer network.

5. CONCLUSIONS

As with the centralized database concurrency control, the algorithms and parameters of the concurrency control for a distributed database are also application and system dependent. In this section the major conclusions on the locking granularity, the algorithms for class 1 and class 2 type transactions are reviewed.

5.1. Locking Granularity

In general, a finer granularity is required for locking in a distributed database than in a centralized database. However, if the locks are well-placed with respect to the accessing transactions, the finest granularity is still not worth the additional concurrency produced.

The need for finer granularity in a distributed database was caused by one major factor: transactions held locks at one node while waiting for locks at another node.

When that condition was avoided with the PS2 model, much coarser granularity was acceptable.

Even that model, however, required slightly less coarse granularity than was required for a centralized database under the same assumptions. In the centralized database, 10 to 100 granules produced the maximum useful computer utilization under the well-placed lock assumptions. In the PS2 distributed database, 100 to 1000 granules are required. In the PS2 model and very coarse granularity, many distributed transactions have to release and rerequest locks at a low number nodes. The additional locking overhead makes coarse granularity unacceptable.

5.2. Class 1 Transactions

If the number of distributed transactions is low ($\leq$10%) and the network is considered lightly loaded, the performance of all four concurrency control algorithms was very similar for class 1 transactions.

As the percentage of distributed transactions increase, the primary site 2 model produces better computer utilization and average response times than either of the decentralized models. In these cases, the extra two messages required in the primary site model represent a lower percentage of overhead since the transactions will be sending at least 2 * NSLAVE messages anyway. Moreover, this overhead is more than offset by the ability to avoid inactive nodes.

When the bandwidth of the network is lowered and the number of distributed transactions is low, however, the decentralized concurrency control models produce better computer utilization and response time than the primary site models. In these cases, the primary site lock message overhead interferes with the normal transaction processing.

The above two conclusions come into conflict as the percentage of distributed transactions increases and a low bandwidth network is assumed. The simulation results indicate that with a low bandwidth network, the SNOOP distributed concurrency control algorithm is best when less than 45% of the transactions are distributed. When more than 45% of the transactions are distributed, the primary site 2 model is preferred.

When the percentage of distributed transactions is less than 10%, the SNOOP and wound-wait algorithms perform equally well. However, as that percentage increases, the SNOOP model results in better performance than the wound-wait model. As expected in these cases, the percentage of conflicts increases and has a more adverse effect on the wound-wait algorithm.

5.3. Class 2 Transactions

Under the class 2 transaction assumption, all of the transactions are small and randomly access entities in the database. In these cases, the decentralized concurrency control models consistently produce better response times and useful I/O and CPU utilization than the primary site models. With extremely small transactions, the extra messages in the primary site models represent a significant delay factor. Furthermore, the small transactions make the probability of conflict and restart very low with the dectralized concurrency control algorithms.

Also, with only small transactions and random lock placement assumptiors, the locking overhead is a significant factor. When all of this overhead is concentrated at one site, that site can bottleneck as either the number of sites in the network or the percentage of distributed transactions increase.

The above observations for class 2 transactions hold even under optimistic network conditions. As the network parameters become restrictive, the advantages of the decentralized concurrency control become even more evident.

The wound-wait and SNOOP concurrency control models produced extremely similar results for class 2 transactions. This similarity was due to two factors. First, the small transactions are involved in very few conflicts and thus the probability of a transaction blocking and being blocked by an older distributed transaction is extremely small. The second factor is that a transaction is much more likely to be blocked by an older transaction (in which case, no wound or kill takes place) since the individual sites operate with a preclaim locking strategy.

CHAPTER 4

CONCLUSIONS

The major goal of this thesis was to examine the effects of concurrency control on the performances of database management systems. The effects of concurrency control on performance are dependent on two conflicting factors. On the other hand, the database system performance can be enhanced by allowing concurrent users simultaneous access to the database. Both the useful computer utilization and the average response time can be improved by supporting a multiple user environment.

On the other hand, the database system performance might be degraded due to extensive concurrency control overhead. The concurrency control overhead is due to the computer resources utilized in some type of "locking". The "locking" is used to prevent one user of the database from interfering with the processing of another user.

In the first section of this chapter, the major conclusions from Chapters 2 and 3 are reviewed. In the next section the applications of these conclusions to other concurrency control implementations are projected and several areas of further research are suggested.

1. SUMMARY OF PREVIOUS CONCLUSIONS

Simulation models were used to study the performance effects of concurrency control in both centralized and distributed databases.

1.1. CENTRALIZED DATABASES

In a centralized database, all database activity, including concurrency control, are processed on a single computer system. A simulation model was used to determine the optimum granularity for locking, the effects of a variety of workload and system characteristics, the effects of a lock hierarchy, and the effects of a "pre-claim" versus a "claim as needed" locking strategy.

The overall conclusions on locking granularity are application dependent as shown in Table 4-1.

Table 4-1 Locking Granularity

	small Transactions	large Transactions	mixed sized Transactions
Well-placed	Coarse gran.	Coarse gran.	Coarse gran.
Random placement	Fine gran.	Coarse gran.	Lock Hierarchy with Fine gran.

In many cases coarse granularity, such as file or relation locking, is preferred. However, if random lock placement is assumed and all of the transactions are small, the coarse granularity is unacceptable and fine granularity locking must be implemented.

If random lock placement is assumed and a variety of different sized transactions are present in the workload, a lock hierarchy should be used. In such a hierarchy, some large transactions can lock large granules, while other small transactions lock much finer granules. If a transaction were to set more than 1% of the smaller locks under any one large lock, it would be more efficient for that transaction to simply set the one large lock.

In a preclaim locking strategy, a transaction acquires all of its locks at the beginning of the transaction. In a claim as needed locking strategy, the locks are acquired as the respective parts of the database need to be accessed. With a few exceptions, the preclaim strategy produced better machine utilization than the claim as needed model. However, the above conclusions on locking granularity and a lock hierarchy hold, regardless of whether a preclaim or claim as needed strategy is used.

1.2. Distributed Databases

In a distributed database, the database activity, including the concurrency control, are processed on several computer systems connected by a network. Four concurrency control algorithms were simulated in order to study their performance effects under a variety of workload and network conditions.

Two of the algorithms simulated involved a centralized concurrency control where locking for the entire database was controlled at one primary site in the network. In the "primary site 1" model, transactions acquire the locks needed at each node or site in some fixed order. If the locks for one node are denied, the "blocked" transaction waits for those locks while holding locks on lower ordered nodes.

In the alternate centralized control model, the "primary site 2" model, the locks needed at each node are again acquired in some fixed order. However, in this case, if the locks for one node are denied, the 'blocked' transaction releases all currently held locks while waiting for access to the locked granules.

The other two algorithms simulated involved a decentralized concurrency control where locking for the portion of the database at each node was controlled at that node.

In the "wound-wait" model, deadlock is prevented by "wounding" any "young" transaction that dares to block an "older" transaction. The wound is transferred to all sites where the wounded transaction is active. If a wounded transaction is blocked at any site by an "older" transaction, the wounded transaction releases its locks at each site and is then restarted.

In the other decentralized control algorithm, deadlocks are resolved by a global deadlock detector, or "SNOOP". If a deadlock exists, a transaction is picked which also releases its locks at each site and is then restarted.

Which model is best in terms of its effect on the distributed database system performance is also application dependent as shown in Table 4-2. Class 1 transactions refer to a workload environment where the locks are assumed to be well-placed with respect to the accessing transactions and that those transactions are of mixed sizes. Class 2 transactions refer to workloads where all of the transactions are small and random placement of locks is assumed.

In some cases, it appears that the concurrency control mechanism is not a significant factor in the database system performance. For class 2 transactions, additional

Table 4-2: Concurrency Control Models

	Class1 Transactions	Class2 Transactions
Fast Net. Most trans. local	Primary Site or Decentralized	Primary Site or Decentralized
Slow Net. Most trans. local	SNOOP	Decentralized
Fast Net. Most trans. non-local	Primary Site 2	Decentralized
Slow Net. Most trans. non-local	Primary Site 2	Primary Site

simulation runs showed that the preference for decentralized concurrency control could be offset by reducing the database load at the primary site. Thus in these cases, the choice of concurrency control algorithm may again not be significant.

For class 1 transactions, when most of the transactions only required local processing and a slower, lower bandwidth network is assumed, the SNOOP algorithm is preferred. In this case, the SNOOP model was favored because of the lower number of messages required.

Also for class 1 transactions, if most of the transactions are non-local or distributed, the primary site 2 model is preferred. The advantage of the primary site 2 model is that only in that model does a transaction release locks at all other nodes while waiting for locks at one node. In the other three models, it is possible for a transaction to hold locks at one node while waiting for locks at another node.

Another factor which favors the primary site 2 model over the decentralized models when most transactions are distributed, is that in those cases, the primary site model no longer produces heavier message traffic.

The distributed database simulations indicated that some of the coarse granularity conclusions for the centralized database do not hold for the distributed database. However, under the well-placed lock assumptions, the finest granularity is still worse than a medium granularity concurrency control.

2. <u>FUTURE DIRECTIONS</u>

The results of the simulation studies suggest several areas for future study. Two such areas would be to extend the lock hierarchy and the claim as needed locking models to a distributed database. Another study would be to investigate the multiple copy problem in the distributed database model. The results of the simulations in this study do, however, provide some insights in each of these areas.

For the centralized database, the conclusion was reached that if the locks are well-placed, coarse granularity is preferred and a lock hierarchy is thus not beneficial. In those cases, it was more efficient to just use one level of coarse locking (10 to 100 locks). In the distributed database cases, finer granularity (500 to 1000 locks) is required even if well-placed locks are assumed. A lock hierarchy in that granularity range was beneficial. Thus a lock hierarchy at each node for a distributed database might be more useful than in a centralized database. This projection could be verified by simple extensions to the distributed database simulations similar to the extension in chapter 2.

A claim as needed locking strategy may be required if the entities to be accessed, and hence the granules to be locked, are dependent upon the values of entities previously accessed. With a claim as needed locking strategy in a distributed database, the primary site models might require two messages for every lock set. In addition, with claim as needed locking, the primary site models would also have to prevent or detect deadlock and thus lose one of their advantages over the decentralized models. Therefore, for claim as needed locking, the primary site models would probably not be acceptable.

The comparison of the two decentralized concurrency control algorithms might be affected by a claim as needed locking strategy. With the preclaim locking strategy and the wound-wait model, relatively few transactions were wounded since there was a high probability that a blocking transaction was older than the blocked transaction. With a claim as needed locking strategy, however, a transaction would request locks at several different instances during its lifetime. Thus, the probability of being blocked by a younger transaction would increase. Consequently, the global deadlock detecter or SNOOP algorithm would probably be better than the wound-wait algorithm in a claim as needed locking environment. Simple simulation extensions could also be used to test that hypothesis.

The multiple copy concurrency problem was discussed in Chapter 1. In a distributed database, it is sometimes advantageous to replicate parts of the database at several of the nodes in the network. The multiple copy concurrency problem is to ensure that the replicated copies are kept mutually consistent or identical during simultaneous user updates.

The four distributed database concurrency control simulations could be applied to the multiple copy problem as follows. Assume that the entire database is replicated at each node. Some transactions are 'read-only' transactions and just need to access the data at one node. These transactions can be considered the local transactions in the simulations. The 'write' transaction, on the other hand, must cause activity at each node and thus may be considered the distributed transactions.

In this interpretation, the PREDIST parameter would represent the percentage of update transactions. Under the above interpretation, the conclusion summarized in Table 4-2 can be applied to the multiple copy problem. If the database is dominated by updates (i.e. most transactions non-local) and the updates are relatively large and sequential in nature (i.e. Class 1 transactions), a primary site concurrency control is suggested. Thus all transactions would first acquire locks at a 'primary copy' of the data.

However, if all of the updates are small and random in nature (i.e. Class 2 transactions) or most of the transactions are 'read-only' with respect to this database portion (i.e. local transactions) then a decentralized concurrency control is suggested (or is at least acceptable). In a decentralized concurrency control, the updates would request locks at each node and proceed with the updates. However, the updates would have to be prepared to be rolled back due to conflicts with other updates.

However, the above analysis is an over simplification of the multiple copy problem in a distributed database. One over simplification is that other concurrency control solutions exist to the multiple copy problems which are not directly extendible to the internal database consistency problems. These algorithms must also be compared with the simulated algorithms.

More importantly, the above analysis assumes a fixed distribution of the copies in the distributed database. In other words, the database is fully replicated and then the number of updates and the network parameters are varied. But the optimum replication of the data actually depends on the proportion of updates and the network parameters. In fact the optimum replication of the data may be influenced by the multiple copy concurrency control.

These analysis deficiencies cannot be over come by straightforward extensions to the existing simulation models. Instead a more complete model should be developed to jointly study the database consistency and multiple copy problems.

In summary, this dissertation provides insights into the effects of concurrency control on database system performance under a wide variety of conditions. The results of the dissertation can be used to guide concurrency control implementations and parameterizations in database management systems.

REFERENCES

ALSB76	Alsberg, P.A., Belford, G.G., Day, J.D. and Grapa, E., "Multi-copy Resiliency Techniques", CAC Doc.202, Center for Advanced Computation, University of Illinois at Urbana-Champaign, May 1976.
ASTR76	Astrahan, M. et.al, "System-R: Relational Approach to Database Management," ACM Transactions on Data Base Systems, Vol.1, No.2, June 1976. pp. 96-137
BERN77	Bernstein, P.A., Shipman, D.W., Rothnie, J.B., and Goodman, N., "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases", Technical Report CCA-77-09, December 1977.
CHAM74	Chamberlin, D. et. al, "A Deadlock-Free Scheme for Resource Locking in a Data Base Environment",IBM Research Report, San Jose, Ca., June, 1974.
CODA71	Data Base Task Group of the CODASYL Programming Language Committee, April, 1971.

CODA73 CODASYL Programming Language Committee. CODASYL COBOL Data Base Facility Proposal, March 1973.

COFF71 Coffman, Jr. E.G., Elphick, M.J., Shoshani, A., "System Deadlocks" Computing Surveys, Vol.3 No.2, June 1971 pp 67-78.

COUR71 Courtois, P.J., Heymans, F., and Parnass, D.L., "Concurrent control with readers and writers", Communications of the ACM, Vol. 14 No.10, October 1971, pp.667-668.

DEC77 Digital Equipment Corporation, "DBMS-11 Data Base Administrator's Guide", DEC-11-ODABA-A-D, 1976.

DIJK69 Dijkstra, E.W., "Cooperating sequential processes", In Programming Languages. F. Genuys, ed., Academic Press, New York, 1968, pp.43-112.

ELLI77 Ellis, C.A., "A Robust Algorithm for Updating Duplicate Databases", Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977, Berkeley, California, pp. 146-158.

EPST78	Epstein, R., Stonebraker, M., and Wong, E., "Distributed Query Processing in a Relational Data Base System", ACM SIGMOD International Conference on Management of Data, Austin, Texas, pp. 169-180.
ESWA76	Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, L. I., "On the Notions of Consistency and Predicate locks in a data base System ", Communications of the ACM, Vol.19, No.11, November, 1976. pp. 624-633.
FLOR74	Florentin, J.J., "Consistency Auditing of Data Bases", The Computer Journal, Vol.17, No.1, February, 1974, pp. 52-58.
GARC78	Garcia-Molina, H., "Performance Comparison of Two Updates Algorithms for Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August 1978, San Francisco, California, pp. 108-119.

GRAP76 Grapa, [illegible] Distri-buted [illegible] Thesis, University [illegible]

GRAY75 Gray, J.N., Lorie, R.A. and Putzolu, G.R. "Granularity of Locks in a Shared Data Base", Proc. 1975 VLDB Conference, Framingham, Mass., Sept., 1975, pp. 428-451.

GRAY76 Gray, J. N., Lorie, R. A., Putzolu, G. R. and Traiger, I. L., "Granularity of Locks and Degrees of Consistency in a Shared Data Base." Proc. IFIP Working Conference on Modelling of Data Base Management Systems, Freudenstadt, Germany, January 1976, pp. 695-723.

GRAY78 Gray, J., "Notes on Data Base Operating Systems", IBM Research Report, RJ 2188, San Jose, California, 1978.

HAWT79 Hawthorne, P. and Stonebraker, M., "The Use of Technological Advances to Enhance Database Management System Performance, University of California, Electronics Research Laboratory, ERL Memo M79/3, January, 1979.

HEWL77 Hewlett-Packard Corporation, "IMAGE Reference Manual", 1977.

KLEI76 Kleinrock, L., "Queuing Systems", Vol.2, John Wiley and Sons, Publisher, 1976.

LAMP78 Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, Vol.21, No.7, July, 1978, pp.558-565.

LBL76 Proceedings of the First Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1976, Berkeley, California.

LBL77 Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977, Berkeley, California.

LBL78 Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August, 1978, San Francisco, California.

LIPS76 Lipson, W. and Lapezak, "LSL User's Manual", Computer Systems Research Group, University of Toronto, Technical Note No.9, August, 1976.

MACR76 Macri, P., "Deadlock Detection and Resolution in a CODASYL Based Data Management System," Proc. 1976 ACM-SIGMOD Conference on Management of data, Washington, D. C., June, 1976 pp. 45-50.

MENA78 Measce, D.A. and Muntz, R.R., "Locking and Deadlock Detection in Distributed Databases", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August, 1978, San Francisco, California, pp. 215-232.

MUNZ77 Munz, R., Krenz G., "Concurrency in Database Systems - A Simulation Study", Proc. ACM SIGMOD International Conference on Management of Data, Toronto, Canada, August, 1977. pp. 111-120.

NAKA75 Nakamura, Yoshida, I. and Hidefumi, K., "A Simulation model for a database system performance evaluation", AFIPS Conference

Proceedings 1975 National Computer Conference, Vol.44, May, 1975, Anaheim, California, pp.459-466.

RIES77	Ries, D. R., Stonebraker, M. "Effects of Locking Granularity in a Database Management System", ACM Transactions on Database Systems, Vol.2, No.3, September, 1977 pp. 233-246.
RIES79	Ries, D.R., Stonebraker, M., "Locking Granularity Revisited", ACM Transactions on Database Systems, Vol.3, No.2, June, 1979.
RODR76	Rodriquez-Rosell, J., "Empirical Data Reference Behavior in Data Base Systems" Computer, Vol.9, No.11, November 1976 pp 9-13.
ROSE77	Rosenkrantz, D.J., Teams, R.E., and Lewis, P.M., "A system Level Concurrency Control for Distributed Database Systems", Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977, Berkeley, California, pp. 132-145.

SPIT76 Spitzer, J. F., "Performance [illegible] of Data Management Applications", Proc. ACM'76 Annual Conference, Houston, Texas, October 1976. pp. 287-297.

STEA76 Stearns, R. E. et al, "Concurrency Control for Data Base Systems", Proc. 1976 IEEE Symposium on Foundations of Computer Science, October 1976. pp. 19-32.

STON74 Stonebraker, M., "High Level Integrity Assurance in Relational Data Base Systems", University of California, Electronics Research Laboratory, Memo ERL-M473, August, 1974.

STON77 Stonebraker, M. and Neuhold, E., "A Distributed Database Version of INGRES", Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977. Berkeley, California, pp. 19-36.

STON78 Stonebraker, M., "Concurrency Control of Multiple Copies of Data in Distributed INGRES", Proceedings of the Third Berkeley Workshop on Distributed Data Management and Computer Networks, August, 1978, San Francisco, California, pp. 235-258.

THOM78 Thomas, R.A., "A Solution to the Update Problem for Multiple Copy Databases which uses Distributed Control", BBN Report 3340, July 1978.

WONG77 Wong, E., "Retrieving Dispersed Data from SDD1: A System for Distributed Databases", Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May, 1977, Berkeley, California. pp. 217-275.

YAO77 Yao S. B., "Approximating Block Accesses in Database Organizations", Communications of the ACM, Vol. 20, No. 4, April 1977, pp 260-261.

DATE
ILMED
-8